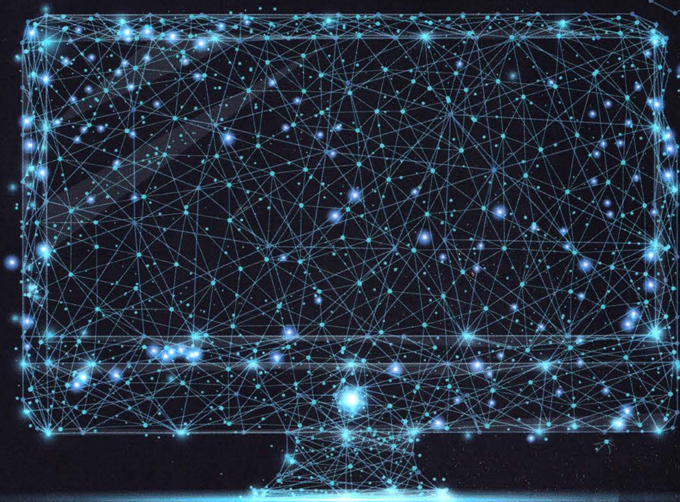
Эта книга — 100%-ная гарантия вашей уверенной работы с языком Python. Освой сам Python "с нуля"!

Кольцов Д. М.



PYTHON

Полное руководство

- #» От базовых основ языка до объектно-ориентированного программирования
- #» Метапрограммирование, многопоточность и масштабирование
- #» Управление программным кодом Python
- #» Наглядные практические примеры

ПОЛНОЕ РУКОВОДСТВО



Кольцов Д. М.

PYTHON

ПОЛНОЕ РУКОВОДСТВО



"Издательство Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973

Кольцов Д. М.

Python. Полное руководство. — СПб.: Издательство Наука и Техника, 2022. — 480 с., ил.

ISBN 978-5-94387-270-9

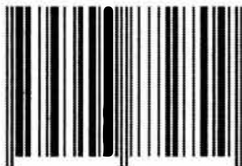
Эта книга поможет вам освоить язык программирования Python практически с нуля, поэтапно, от простого к сложному. Первая часть книги посвящена базовым основам языка: переменные и типы данных, операторы, циклы и условные операторы, математические функции, кортежи, множества и словари, итераторы и генераторы, модули и пакеты, а также многое другое.

Во второй части книги перейдем к более сложным вещам в Python: объектно-ориентированное программирование, метапрограммирование, многопоточность и масштабирование.

Отдельное внимание будет уделено документированию своего проекта в Python, контролю и оптимизации кода. Теоретическая часть книги сопровождается практическими примерами, позволяющими на практике осваивать полученные теоретические знания.

Книга будет полезна как начинающим, так и тем, кто хочет улучшить свои навыки программирования на Python.

ISBN 978-5-94387-270-9



Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Вкратце о Python

Python – одновременно мощный и простой язык программирования, разработанный Гвидо ванн Россумом (Guido van Rossum) в 1991 году. С одной стороны, язык довольно молодой (тот же С был разработан в 1972 году), с другой стороны, ему уже 30 лет и за это время его успели довести до того уровня, когда на нем можно написать проект любого масштаба, в том числе коммерческие приложения, работающие с очень важными данными.

Python – это высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Другими словами, писать программы на Python довольно просто, а код самих программ будет гораздо лучше восприниматься, чем в случае с другими языками программирования.

Синтаксис ядра этого языка программирования минималистичен, однако, стандартная библиотека содержит множество полезных функций.

Тех вольностей, которые себе программист мог позволить в других языках программирования, например, в Pascal, С, Java или PHP, здесь непозволительны. Один лишний или недостающий отступ – и вы получите синтаксическую ошибку. Так Python приучает программиста писать красивый и читабельный код. С одной стороны, если я бы не советовал выбирать Python в качестве первого языка программирования – вам будет сложно. С другой

стороны, это как учиться ездить на автомобиле зимой в гололед. После такой школы летом вы уж точно будете ездить – с легкостью сможете освоить другие языки программирования.

Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное. Это означает, что вы можете выбрать любой стиль программирования. Новичкам, создающим относительно несложные программы, подойдет функциональный стиль. А для серьезных проектов, как правило, выбирают объектно-ориентированное и/или аспектно-ориентированное программирование.

Кросс-платформенность

Python портирован и работает почти на всех известных платформах – от карманных компьютеров и смартфонов до мейнфреймов. Существуют версии Python под Microsoft Windows, практически все варианты UNIX (включая FreeBSD и Linux), Plan 9, macOS и macOS X, iPhone OS 2.0 и выше, Palm OS, OS/2, Amiga, HaikuOS, AS/400 и даже OS/390, Windows Mobile, Symbian и Android.

Важно понимать, что программы, написанные на Python, независимы от платформы. Другими словами, вы можете написать программу, которая будет работать и на вашем macOS X и на Windows-компьютере приятеля и даже на iPhone. Главное, чтобы на устройстве, на котором планируется запускать программу, был установлен Python.

Python – один из самых простых языков программирования

Когда-то давно программы писались с помощью переключения различных разъемов на компьютерах, которые занимали целые залы. Затем появились перфокарты – специальные карточки, используемые для ввода программы, после – язык программирования Ассемблер, позволяющий манипулировать данными прямо в регистрах процессора. Это пример низкоуровнего языка. Даже самая простая программа на этом языке представляла довольно большой, сложный и непонятный непосвященному человеку кусок кода.

С появлением высокоуровневых языков, таких как C, Java все изменилось. Такие языки программирования ближе к человеческому языку, чем к машинному. Python делает синтаксис еще проще. Его правила приближают-

ся к английскому языку. Создание программ на Python настолько простой процесс, что о нем говорят как о "программировании со скоростью мысли". Все это позволяет повысить производительность труда программиста – программы на Python требуют меньше времени на разработку, чем программы на других языках программирования.

Популярность Python

Если вы раньше ничего не слышали о Python и думаете, что он непопулярен, то вы ошибаетесь. Его используют разработчики со всего мира, ним пользуются крупнейшие корпорации, такие как Google, NASA, Red Hat, Yahoo!, Xerox, IBM, Microsoft и др. Так, Google предпочитает C++, Java и Python[1], а Microsoft даже открыла Python Developer Center[2] .

Популярные программные продукты Yahoo, в том числе Django[3], TurboGears[4] и Zope[5] написаны на Python.

Некоторым начинающим программистам, которые только выбирают язык программирования, программы на Python могут показаться неказистыми. Порой кажется, что ничего серьезного на этом языке не разработаешь, и он больше приближен к сценариям оболочки, чем к полноценным языкам программирования, таким как C#.

Это заблуждение. Многие популярные игры от EA Games, 2K Games и Disney Interactive, написаны на Python:

https://gamedev.stackexchange.com/questions/5035/famous-games-written-in-python

Игра – это одно из самых сложных приложений, поскольку оно сочетает работу с графикой, музыкой, сложную логику и т.д.

Сколько это стоит?

Интерпретатор Python абсолютно бесплатен и распространяется свободно. Чтобы использовать его, вам не нужно ничего платить, но зато вы можете абсолютно свободно продавать свои программы, написанные на Python – это позволяет лицензия, по которой распространяется этот интерпретатор.

1 https://www.quora.com/Which-programming-languages-does-Google-use-internally
2 https://azure.microsoft.com/en-us/develop/python/

3 https://en.wikipedia.org/wiki/Django_(web_framework)
4 https://en.wikipedia.org/wiki/TurboGears
5 https://ru.wikipedia.org/wiki/Zope

Такие условия лицензии способствуют популярности этого языка программирования. Ведь для старта вам не нужно ничего и никому платить. Вы можете использовать Python, как для обучения программированию, так и для создания коммерческих программ. В отличие от других языков программирования, где за использование среды программирования нужно выложить кругленькую сумму, например, за Visual Studio придется выложить более 500 евро, а стоимость некоторых сторонних компонентов превышает 1500 евро (например, DevExpress).

Понятно, что это останавливает многих программистов-одиночек или заставляет их использовать пиратские версии. А на Python вы можете программировать с чистой совестью – все изначально доступно бесплатно.

Философия Python

Чтобы введение не было однотипным и скучным, расскажу вам о философии Python. Python – это не простой язык программирования, у него есть своя философия, разработанная Тимом Петерсом.

Философия выводится один раз за сессию при вводе команды

```
import this
```

Результат выполнения этой команды изображен на рис. В.1.

```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
```

Рис. В.1. Философия Python

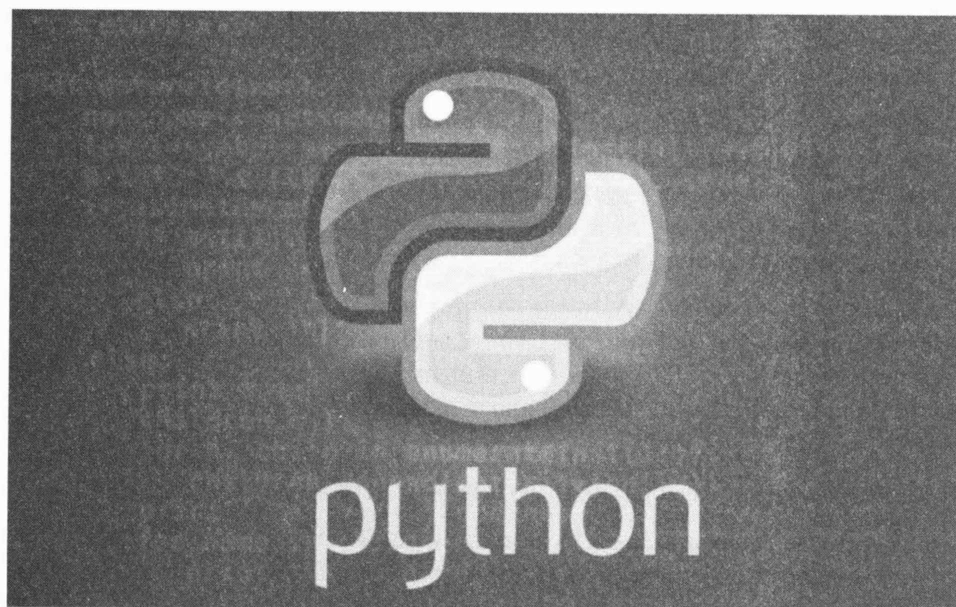
Вот текст философии в переводе с английского языка:

```
Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Плоское лучше, чем вложенное.
Разреженное лучше, чем плотное.
Читаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
При этом практичность важнее безупречности.
Ошибки никогда не должны замалчиваться.
Если не замалчиваются явно.
Встретив двусмысленность, отбрось искушение угадать.
Должен существовать один — и, желательно, только один —
очевидный способ сделать это.
Хотя он поначалу может быть и не очевиден, если вы не
голландец.
Сейчас лучше, чем никогда.
Хотя никогда зачастую лучше, чем прямо сейчас.
Если реализацию сложно объяснить — идея плоха.
Если реализацию легко объяснить — идея, возможно, хороша.
Пространства имён — отличная штука! Будем делать их побольше!
```

Приятного чтения!

ГЛАВА 1.

ОСНОВЫ. ПЕРВАЯ ПРОГРАММА



1.1. О версии Python

На момент написания этих строк текущей является версия 3.9.2. Однако параллельно с веткой 3.х доступна и ветка 2.7 (хотя ее поддержка завершена в прошлом, 2020-ом, году). Какую версию выбрать – 3.х или 2.7?

В Интернете вы можете часто найти рекомендации бывалых программистов, рекомендующих использовать версии 2.7. Они мотивируют это наличием огромного количества кода, написанного под версию 2.7, своим опытом разработки в версии 2.7 и, конечно же, нежеланием переходить на ветку 3.0 (тогда придется переписать много кода).

Однако посмотрите на дату, когда были написаны эти рекомендации. Скорее всего, это будет 2009 или 2012 год максимум. Первая версия Python 3.0 появилась в декабре 2008 года, с тех пор уже много воды утекло, и большая часть их рекомендаций уже потеряла свою актуальность.

Если вы – начинающий программист и только хотите начать осваивать Python, вам следует выбрать самую новую версию – 3.9.

Совсем другое дело, когда вы – уже состоявшийся программист, знаете другие языки программирования (например, С++, PHP) и вам необходимо изучить Python, поскольку вам предложили место в компании, которая ведет разработку на этом языке программирования, тогда все зависит от компании.

Если компания использует версию 2.7, тогда выбора у вас нет – его за вас сделала компания, а переписывать весь код на 3.x – нет времени. Подробно о разнице между версиями 2.7 и 3.0 рассказано здесь:

https://wiki.python.org/moin/Python2orPython3ё

Во всех остальных случаях нет смысла говорить о ветке 2.7, так как ее поддержка завершена и больше нет смысла ее использовать.

1.2. Установка Python 3

Как вы уже догадались, далее будет рассматриваться только ветка 3.x как самая актуальная на данный момент. Скачать инсталлятор Python можно по адресу:

https://www.python.org/downloads/

Обратите внимание, что Python – это кросс-платформенный язык, и есть его версии для Linux и MacOS. Далее все иллюстрации будут соответствовать Windows 10, поскольку эта ОС используется большинством пользователей в данный момент.

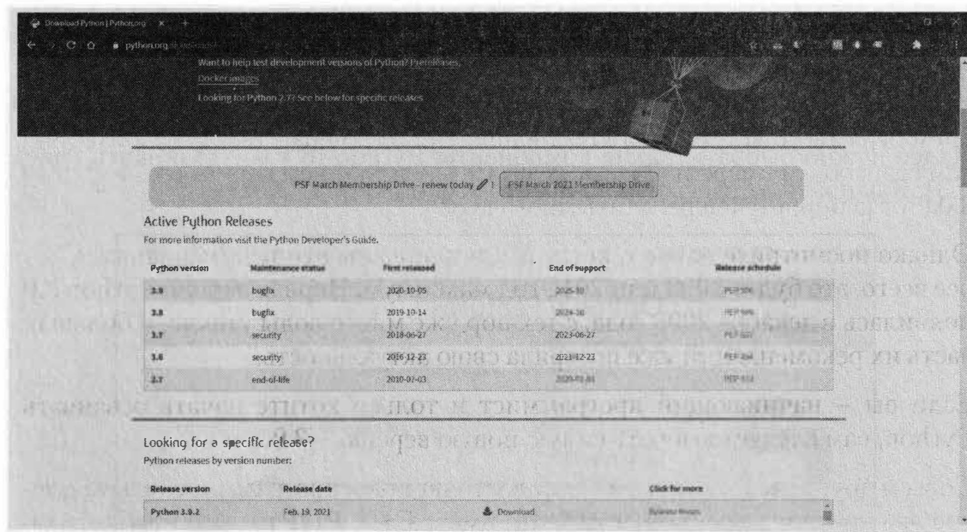


Рис. 1.1. Страница загрузки Python

Скачайте и запустите инсталлятор (файл python-3.9.2amd64.exe). Первым делом нужно выбрать определиться, что вы хотите сделать – или установить с установками по умолчанию (кнопка **Install Now**) или кастомизировать инсталляцию. Стандартные установки использовать не рекомендуется хотя бы по той причине, что Python 3.9.2 устанавливается в домашний каталог пользователя, а обычно нужно сократить путь к интерпретатору, поэтому лучше выбрать **Customize installation**, а перед этим включить флажок **Add Python 3.9 to PATH**, чтобы инсталлятор сам добавил путь к интерпретатору в переменную PATH.

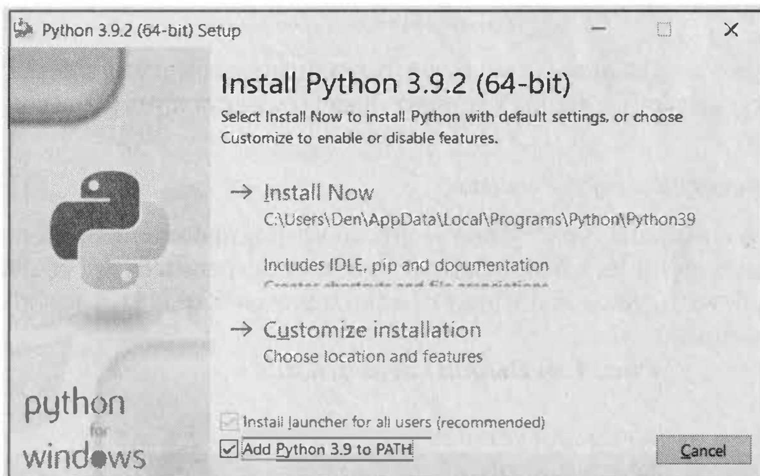


Рис. 1.2. Тип установки

Далее нужно выбрать, какие компоненты Python нужно установить (рис. 1.3). Как правило, здесь нужно просто нажать кнопку **Next**.

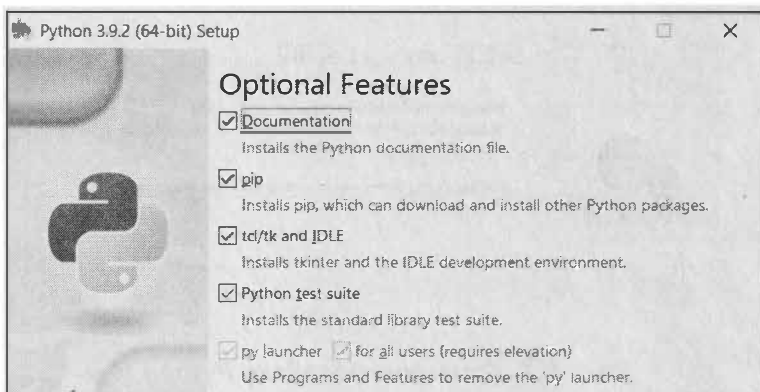


Рис. 1.3. Выбор компонентов Python

Затем нужно выбрать каталог для установки (рис. 1.4). Как уже было отмечено, по умолчанию Python устанавливается в домашний каталог пользователя, вызвавшего установку, но лучше установить его в корневой каталог (например, в C:\Python) – там вам будет удобнее работать с интерпретатором.

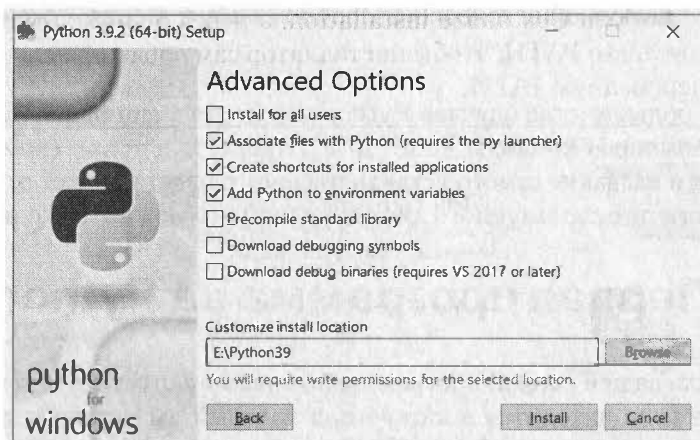


Рис. 1.4. Выбор каталога для установки

При выборе каталога для установки есть возможность выбрать различные опции вроде ассоциации файлов .py с Python, добавления Python в переменные окружения и т.д. Установите опции по своему усмотрению.

После нажатия кнопки **Install** начнется процесс установки Python и нужно будет немного подождать. По окончанию этого процесса нужно нажать кнопку **Close** (рис. 1.5).



Рис. 1.5. Python 3.9.2 установлен

Поздравляю! Вы только что установили Python и можете приступить к его изучению.

> Технические подробности. При установке инсталлятор связывает расширение .py с программой python.exe, а расширение .pyw с программой pythonw.exe. Первая программа используется для выполнения консольных приложений, а вторая – оконных.

В Linux в большинстве случаев Python уже будет установлен. Если это не так, то с помощью команды `sudo apt install python` (команда установки, как и название самого устанавливаемого пакета, может отличаться в зависимости от используемого дистрибутива) это можно легко исправить.

1.3. Первая программа на Python

Для запуска вашей первой программы вы можете запустить или программу python.exe (она находится в каталоге, в который вы установили Python[1]) или запустить среду разработки IDLE (это можно сделать с помощью меню **Пуск**) – см. рис. 1.7.

1 Если вы добавили Python в PATH, то можно просто ввести команду python

Рис. 1.6. Программа python.exe

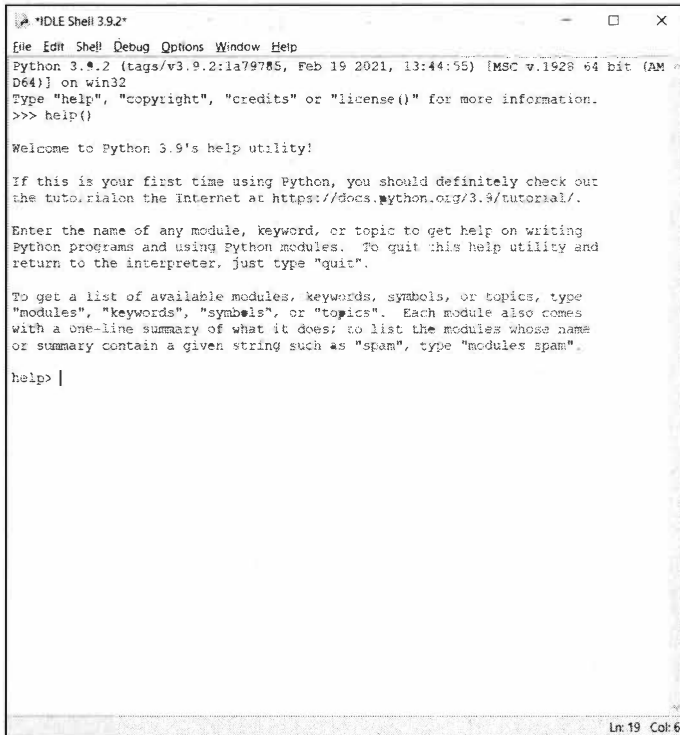


Рис. 1.7. Оболочка IDLE

Начинающих программистов вид командной строки наверняка отпугнет, поэтому лучше выбрать IDLE.

Код вашей первой программы представлен в листинге 1.1. Не нужно быть великим программистом, чтобы догадаться, что делает эта программа. Во многих учебниках по программированию первая программа обязательно должна выводить строчку Hello, world! Мы не стали отступать от традиции и решили написать такую же программу.

Листинг 1.1. Первая программа

```
# Comment
print("Hello, world!")
```

Как вводить программу? Вводите программу строчка за строчкой, а по окончанию ввода строки нажимайте **Enter**. Поскольку Python – это интерпретатор, то результат выполнения строки кода вы увидите мгновенно. В результате обработки комментария Python, как и следовало бы ожидать,

ничего не выведет. А вот в результате обработки второй строчки кода будет выведен текст Hello, world! Результат выполнения сценария изображен на рис. 1.8.

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello, world!")
Hello, world!
>>> |
```

Рис. 1.8. Результат выполнения сценария. Первая программа

1.4. Подробно о IDLE

1.4.1. ПОДСКАЗКИ ПРИ ВВОДЕ КОДА

Несмотря на кажущуюся простоту IDLE оснащена функцией подсказки при вводе кода – как у "полноценных" сред программирования (рис. 1.9). Введите название функции или метода, и вы получите подсказку по параметрам, которые нужно передать этой функции или методу.

```
*IDLE Shell 3.9.2*
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello, world!")
Hello, world!
>>> print(
          print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Рис. 1.9. Подсказка при вводе кода

Введите print("Hello") и нажмите **Enter**. Вы только что написали свою первую программу и получили результат ее выполнения – строчку "Hello".

Функция print() выводит на консоль текст, помещенный вовнутрь скобок и заключенный в кавычки. Если в скобках ничего нет, то будет выведена пустая строка.

> Внимание! Python чувствителен к регистру символов. Все названия функций пишутся строчными буквами, поэтому print() – это правильно, а Print() или PRINT() – нет.

1.4.2. ПОДСВЕТКА СИНТАКСИСА

Среда IDLE, как и ее старшие собратья, обеспечивает подсветку синтаксиса. Это означает, что слова на экране отображаются разными цветами. Такое явление упрощает понимание того, что именно вы вводите. Каждое слово по определенному правилу окрашивается в определенный цвет.

Так, имена функций окрашиваются в фиолетовый цвет, строковые значения – в зеленый. Если IDLE не может распознать, что именно вы ввели, то это слово никак не окрашивается. При светлой теме оформления оно будет напечатано черным, при темной – белым.

Результат работы программы интерпретатор выводит на экран шрифтом голубого цвета – так вы можете отделить код от его результата визуально.

Если вы проделали пример с функциями print() и Print(), то заметили, что первая функция была окрашена в фиолетовый, а вторая – никак не окрашена. Подсветка синтаксиса позволяет помочь понять, что вы что-то делаете не так и исправить ошибку еще до запуска кода. На первых порах подсветка синтаксиса станет вашим незаменимым помощником, ведь она делает код понятным с первого взгляда и позволяет сразу увидеть ошибки, если таковые имеются.

1.4.3. ИЗМЕНЕНИЕ ЦВЕТОВОЙ ТЕМЫ

Среда IDLE позволяет изменять цвет оформления элементов. Выбрать цветовую тему можно в настройках: **Options, Configure IDLE**. Далее нужно перейти на вкладку **Highlighting** и выбрать тему оформления. На рис. 1.10 показано, что выбрана тема IDLE Dark.

При желании, вы можете создать собственную подсветку. Для этого выберите элемент, для которого вы хотите изменить цвет, а затем нажмите

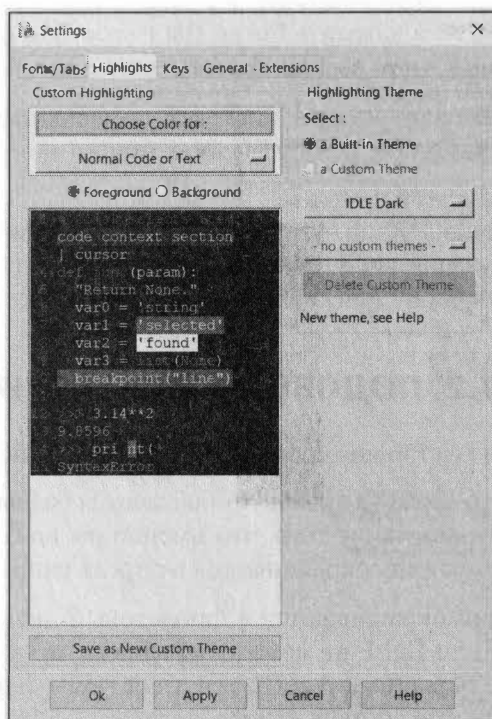


Рис. 1.10. Выбор темы оформления

кнопку **Choose Colour for**. Также можно воспользоваться переключателями **Foreground** (цвет шрифта) и **Background** (цвет фона).

Как только результат вам понравится, нажмите кнопку **Save as New Custom Theme**. Затем включите в группе **Highlighting Theme** переключатель **a Custom Theme** и выберите свою тему оформления.

На вкладке **Fonts/Tab** можно установить другие параметры шрифта – выбрать другую гарнитуру, установить другой размер. По умолчанию используется шрифт Courier New размером 10 пунктов.

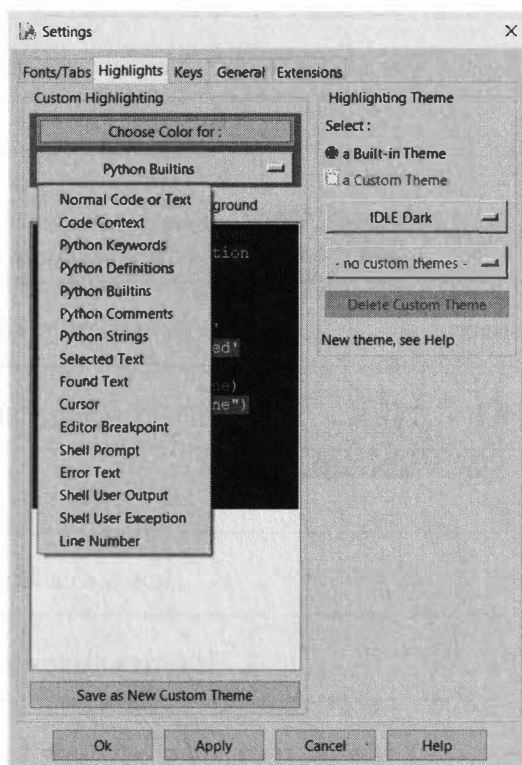


Рис. 1.11. Изменение цветовой темы

1.4.4. ГОРЯЧИЕ КЛАВИШИ

Среда IDLE поддерживает горячие клавиши, приведенные в таблице 1.1.

Таблица 1.1. Некоторые горячие клавиши

Комбинация клавиш	Описание
Home	Переход к началу строки
Ctrl + I	Вставка по центру
Alt + U	Изменение отступа
Alt + X	Проверка модуля

Ctrl + Q	Закрыть все окна
Alt + F4	Закрыть окно (выход)
Ctrl + C, Ctrl + X, Ctrl + V	Стандартные клавиши управления буфером обмена
Ctrl + Backspace	Удалить слово слева
Ctrl + Delete	Удалить слово справа
Ctrl + F	Поиск
Alt + F3	Поиск в файлах
Ctrl + F3	Найти в выделенном
Ctrl + G	Найти снова
Alt + q	Форматировать параграф
Alt + g	Перейти к строке
Ctrl + C	Прервать выполнение (когда оно запущено)
Alt + m	Открыть модуль
Ctrl + N	Открыть новое окно
Ctrl + O	Открыть окно с файлом
Ctrl + P	Распечатать окно

Ctrl + Z	Отменить последнее действие
Esc	Снять выделение
Ctrl + H	Замена в тексте
F5	Запустить модуль на выполнение
Ctrl + F6	Перезапустить оболочку
Alt + Shift + S	Сохранить копию окна как файл
Ctrl + Shift + S	Сохранить окно как файл
Ctrl + A	Выделить все
Tab	Сделать отступ
F6	Просмотреть перезапуск

Внимание! Комбинации клавиш не работают? Обратите внимание на язык ввода. Он должен быть английским. Если вы нажимаете Ctrl + V, а язык ввода – болгарский, то Python видит комбинацию клавиш Ctrl + M и поэтому она не срабатывает! Подход не очень правильный, но какой есть. Именно поэтому я предпочитаю использовать сторонний редактор для редактирования Python-программ, а не редактор среды IDLE. В последний можно загрузить уже готовую программу, например, для ее отладки.

1.5. Помещение программы в отдельный файл. Кодировка текста

Режим интерпретатора хорош, когда вы только учитесь программирования и то – для совсем небольших программ. Когда программа содержит несколько десятков строк кода, лучше поместить ее в отдельный файл с расширением .py.

Кодировка программ, написанных на Python – UTF-8, поэтому для редактирования таких файлов подходит не каждый редактор. Если вы ищете редактор попроще, то можно использовать программу Notepad2 (*http://www.flos-freeware.ch/*). Это базовый редактор, но он поддерживает подсветку синтаксиса Python и кодировку UTF-8. Минимальный набор. Если же вам нужны такие "плюшки", как автодополнение кода, поддержка системы контроля версий Git, можно использовать редакторы вроде Atom или Microsoft Visual Studio Code. Такие редакторы подойдут для даже сложных проектов, состоящих из множества Python-файлов.

Создайте файл с расширением .py. Пусть это будет файл E:\Python\samples\1.py. Выберите кодировку файла в редакторе Atom. Поместите в него всего одну строчку кода:

```
print ("Hello")
```

Сохраните файл. Теперь разберемся, как его запустить. Самый простой способ – открыть командую строку и ввести команду:

```
python E:\Python\samples\1.py
```

> ***Примечание.*** Вам интересно, как я изменил заголовок командной строки в Windows? Для этого используется команда title <ваша_строка>. Для дополнительной информации обратитесь к руководству по командам cmd.exe.

Если вы предпочитаете использовать IDLE, выберите команду меню **File, Open**. Выберите ранее сохраненный файл 1.py. Он откроется в отдельном окне. В этом окне нужно выбрать команду меню **Run, Run Module** или просто нажать F5.

Если запускаемый сценарий писали не вы и его кодировка отличается от UTF-8, ее можно указать так:

```
# - *- coding: cp1251 -*-
```

Вместо cp1251 нужно указать вашу кодировку. Однако можно использовать редактор Notepad2, чтобы просто перекодировать файл в UTF-8. Для этого нужно выполнить команду меню **File, Encoding, Recode** и в появившемся окне выбрать нужную кодировку и нажать **OK**.

1.6. Структура программы

Структура Python-программы довольно проста, но, тем не менее, она отличается от структуры программ, используемой в других языках программирования. Если вы ранее программировали на C/C++, PHP или Java, вам поначалу будет немного непривычно.

Если вы программируете в Linux, то первой строкой должна быть строка, указывающая путь к интерпретатору Python:

```
#!/usr/bin/python
```

Обратите внимание: пробелов быть не должно. Ни до решетки (#), ни после нее, ни после восклицательного знака. Узнать путь к Python в вашей системе можно с помощью команды **which**:

```
$ which python
/usr/bin/python
```

Если вы указали путь к интерпретатору, то вы можете превратить свою программу в полноценный сценарий и запускать ее без указания *python* в командной строке. Сейчас поясню. Пусть у вас есть сценарий first.py. В нем в качестве первой строки указан путь к Python. Вам нужно его сделать исполнимым:

```
$ chmod +x first.py
```

После этого вы можете запустить его так:

```
$ ./first.py
```

Строка с указанием пути к интерпретатору – это не инструкция Python, а инструкция командного интерпретатора **bash**, который, благодаря ней, будет знать, какая программа будет обрабатывать сценарий. Если вы не укажите эту первую строку, **bash** посчитает сценарий first.py собственным сце-

нарием, а поскольку синтаксис **bash** ни разу не похож на синтаксис Python, возникнет ошибка.

Если вы не хотите превращать свои программы в отдельные сценарии, вы можете запускать их, как в Windows:

```
$ python <имя_программы.py>
```

Вторая строка программы (или первая, если вы программируете в Windows) – это строка с указанием кодировки:

```
# -*- coding: utf-8 -*import
```

По умолчанию используется кодировка UTF-8, и эту строку можно было бы не указать, но ее наличие в ваших программах свидетельствует о хорошем тоне. Если кодировка файла отличается от UTF-8, данная строка (с указанием конкретной кодировки) обязательна!

После этих двух служебных строк начинается код самой программы. Примечательная особенность Python заключается в том, что каждая инструкция располагается на отдельной строке и при этом, если она не является вложенной, то должна начинаться с самого начала строки. Рассмотрим пример, продемонстрированный на рис. 1.12 Простейшая программа – и с ошибкой. Как видите, инструкция начинается с пробела, а не с начала строки, поэтому вы получили ошибку

```
SyntaxError: unexpected indent
```

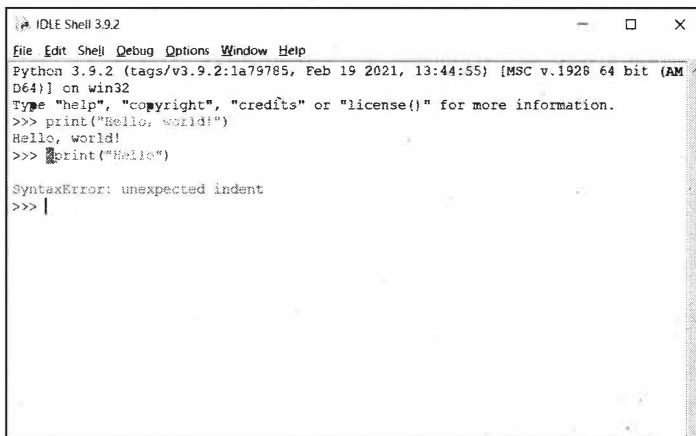


Рис. 1.12. Ошибка: неожиданный отступ

Хорошо, что интерпретатор подсказывает, что не так, и даже показывает позицию ошибки.

Во многих языках программирования (PHP, C, Pascal, Perl, Java и др.) каждая инструкция должна завершаться точкой с запятой. В Python точка с запятой необязательна. Но и ее наличие не вызовет ошибку, если вы вдруг поставили ее по привычке. Поэтому следующие два варианта инструкции – правильные с точки зрения синтаксиса:

```
print("Hi!")
print("Hi!");
```

Концом инструкции в Python считается конец строки (символ EOL). Однако точка с запятой обязательна, если вы хотите поместить в одну строку несколько инструкций, например:

```
>>> a = 2; b = 3; c = 4; x = a + b * c;
>>> print(x)
14
```

Также в других языках программирования мы привыкли видеть фигурные скобки, которые разграничивают инструкции внутри блока. Например, вот код на PHP:

```
$x = 0;
while ($x < 5) {
  echo "$x \n";
  $x++;
}
echo "Все";
```

А вот аналогичный код на Python:

```
x = 0
while x < 5:
    print(x)
    x += 1
print("Все")
```

Перед всеми инструкциями блока должно быть расположено одинаковое количество пробелов. Так Python распознает, какая инструкция и к какому блоку относится. При написании кода в IDLE одинаковое количество пробелов проставляется автоматически, а для завершения блока при переходе на следующую строку вы должны нажать **Backspace**, а затем – **Enter**. При написании кода в редакторе количество пробелов нужно учитывать самостоятельно. Обычно используется 4 пробела. Если количество пробелов внутри блока – разное, Python выведет фатальную ошибку и выполнение программы будет остановлено. Сначала вам будет непривычно, но спустя месяц программирования на Python вы научитесь писать понятный и красивый код. Ведь в других языках программирования вы можете написать хоть все инструкции программы, в том числе вложенные, в одну строку. Главное, чтобы было правильно с точки зрения синтаксиса. В результате читать такую "кашу" из кода не очень удобно. В Python такого быть не может – хочешь не хочешь, а придется создавать понятный код.

Если весь ваш блок состоит всего из одной инструкции, разрешается разместить ее на одной строке с основной инструкцией, например:

```
for x in range (1 , 10): print(x)
```

Если инструкция слишком длинная, то вы можете разбить ее или символом перевода строки (\) или поместив ее в круглые скобки. Во втором случае внутри инструкции вы сможете использовать даже комментарии. Примеры:

```
x = a + b \
       * c

x = (a + b     # Comment
       * c)
```

В первом случае никакие другие символы не разрешаются, в том числе и комментарии.

1.7. Комментарии

Как вы уже успели понять, комментарии в Python начинаются с решетки. Комментарий может начинаться, как с новой строки, так и помещен после инструкции:

```
# Это комментарий
print("Привет")  # Это тоже комментарий
```

Если решетка размещена перед инструкцией, то инструкция считается комментарием и не будет выполнена:

```
# print("Привет")
```

Также # не считается символом комментария, если он находится внутри кавычек или апострофов, например:

```
print("# Комментарий")
```

В Python нет многострочного комментария, поэтому можете или использовать несколько комментариев:

```
# Многострочный
# комментарий
```

Можно также использовать тройные кавычки:

```
"""
Многострочный
комментарий
"""
```

Данная конструкция не игнорируется интерпретатором. Он создает строковой объект, но так как все инструкции внутри тройных кавычек считаются текстом, никакие действия производиться не будут.

Однако знайте, что при частом использовании такого подхода в большой программе будет наблюдаться нерациональное использование памяти, поскольку обычные комментарии игнорируются, а в случае с тройными кавычками создается строковой объект.

С одной стороны, объемом оперативной памяти в 16 Гб сегодня никого не удивишь, и те несколько килобайтов перерасхода памяти в очень большой программе особой роли не сыграют. С другой стороны, нужно привыкать к рациональному использованию ресурсов компьютера. Сегодня вы использовали тройные кавычки, чтобы закомментировать несколько килобайтов текста, завтра – забудете закрыть соединение или освободить память. Ин-

терпретатор Python по окончанию работы сценария автоматически закрывает все выделенные программе ресурсы (в том числе файлы и соединения), но такой подход считается очень плохим тоном среди программистов.

Тройные кавычки удобно использовать, чтобы временно закомментировать какой-то участок кода (чтобы не дописывать в начале каждой строки #, а потом не удалять ее, когда вам вновь понадобится этот фрагмент кода).

Использовать или нет тройные кавычки – решать вам. Я вам показал, как их можно использовать и рассказал о преимуществах и недостатках этого способа.

1.8. Ввод/вывод данных

В этом разделе мы поговорим о вводе и выводе данных. Ранее было показано, что для вывода данных используется инструкция (функция) print(). Полный синтаксис *print()* выглядит так:

```
print ( [<Объекты>][, sep=' '][,end='\n'][,file=sys.stdout])
```

Разберемся с параметрами функции. Первый параметр – это набор объектов, которые нужно вывести. Список объектов разделяется запятыми. Например:

```
>>> a = 1; b = 2;
>>> print(a, b)
1 2
```

Как видите, между объектами автоматически вставляется разделитель – по умолчанию Пробел. Задать собственный разделитель можно с помощью параметра **sep**. Например, вы можете задать символ табуляции:

```
>>> print(a, b, sep='\t')
1  2
```

Параметр **end** задает конец строки. По умолчанию использует символ '\n', который в Windows автоматически преобразуется в последовательность '\r\n' (перевод каретки и новая строка). Обычно вам не нужно изменять этот параметр при выводе на экран, но может понадобиться его изменение при выводе в файл – все зависит от синтаксиса (формата) файла.

Последний параметр задает файл, в который осуществляется вывод. По умолчанию вывод осуществляется в файл sys.stdout, что означает стандартный вывод и обычно соответствует экрану (консоли). О работе с файлами мы пока не говорим, просто знайте, что функция print() умеет выводить данные не только на экран, но и в файл.

Вызов функции *print()* без параметров позволит просто перевести строку (выводится пустая строка):

```
print()
```

Некоторые программисты вместо функции *print()* предпочитают использовать метод *write* объекта sys.stdout. Например:

```
import sys;
sys.stdout.write("Привет")
```

При первом использовании метода *write()* нужно сначала импортировать модуль **sys**, в котором определен этот метод.

Особенность этого метода в том, что он не добавляет символ конца строки, поэтому его нужно при необходимости добавить самостоятельно:

```
sys.stdout.write("Hello\n")
```

Для ввода данных в Python 3 используется функция input(). Использовать ее можно так:

```
[<Переменная> = ] input ( [ <Сообщение>] )
```

Небольшой пример:

```
name = input("Как тебя зовут? ")
```

```
print("Привет, ", name)
```

Работа с программой в IDLE:

```
>>> name = input("Как тебя зовут? ")
Как тебя зовут? Марк
>>> print("Привет, ", name)
Привет, Марк
>>>
```

Обратите внимание, что функция *input()* не выводит после сообщения-приглашения никаких символов, поэтому ввод начнется сразу после выведенного функцией сообщения, что не очень удобно. Поэтому в конец сообщения принято добавлять или пробел или символ новой строки, чтобы ввод начался с новой строки:

```
>>> name = input("Как тебя зовут?\n")
Как тебя зовут?
Марк
```

Если пользователь нажмет Ctrl + Z или будет достигнут конец файла (в

данном случае речь идет о файле стандартного ввода – *stdin*), будет сгенерировано исключение EOFError и программа завершит свою работу. Чтобы этого не произошло, нужно произвести обработку этого исключения:

```
try:
    name = input("Как тебя зовут? ")
    print(name)
except EOFError:
    print("EOFError raised")
```

Подробно обработка исключений рассмотрена не будет, а пока вам нужно знать, как обработать только одно из них – EOFError.

1.9. Чтение параметров командной строки

Вашей Python-программе, как и любой другой программе, можно передать параметры командной строки. Они хранятся в списке **argv** модуля **sys**. Вот как можно вывести все переданные программе параметры:

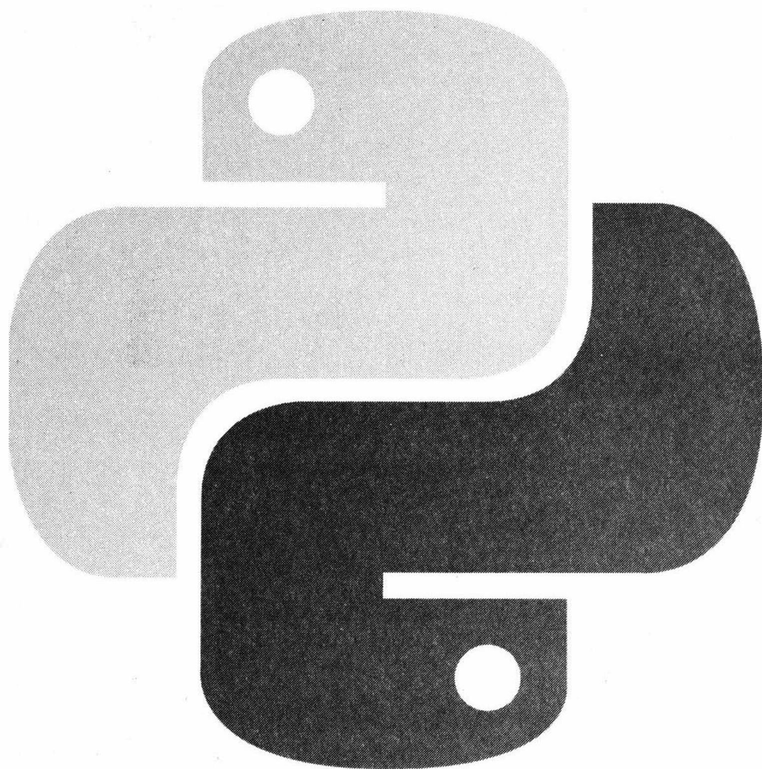
```
import sys
args = sys.argv[:]
for n in args:
    print(n)
```

Передать параметры программе можно так:

```
python program.py arg1 arg2
```

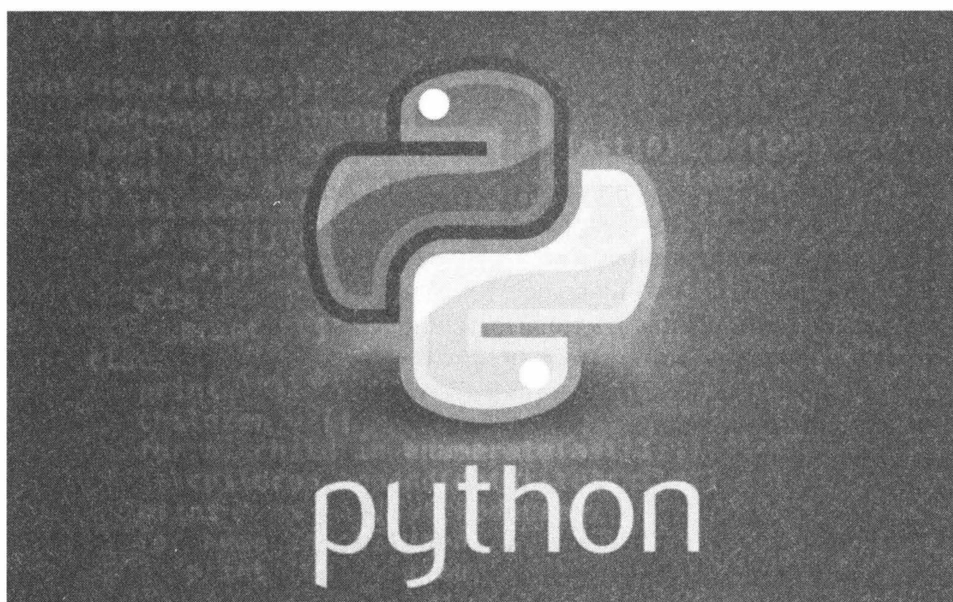
В данном случае будет запущен интерпретатор Python, который начнет обработку программы program.py, а самой программе при этом будут переданы параметры arg1 и arg2.

На этом все. В следующих главах будут рассмотрены переменные и типы данных в Python.



ГЛАВА 2.

ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ



Теперь, когда вы знаете, как создать и запустить программу, можно приступить к изучению синтаксиса языка. Прежде, чем мы перейдем к рассмотрению переменных, вы должны знать, что типизация в Python динамическая, то есть тип переменной определяется только во время выполнения. Данное отличие может поначалу сбивать с толку, особенно, если вы программировали на языках, где требуется сначала указать тип переменной, а потом уже присваивать ей значение (C, Pascal и др.). В то же время языки с динамической типизацией тоже не редкость – типичный пример PHP, где не нужно объявлять тип переменной перед присвоением значения. Тип будет определен автоматически – по присвоенному значению. С одной стороны, так проще. С другой – это требует от программиста постоянно следить за данными, которые он присваивает переменной, поскольку одна и та же переменная в разные моменты времени может содержать данные разных типов.

В Python имеются встроенные типы: булевый, строка, Unicode-строка, целое число произвольной точности, число с плавающей запятой, комплексное число и некоторые др. Из коллекций в Python встроены: список, кортеж (неизменяемый список), словарь, множество и др. *Все значения являются объектами, в том числе функции, методы, модули, классы.*

Все объекты делятся на ссылочные и атомарные. К атомарным относятся **int**, **long** (в версии Python 3 любое число является **int**, так как, начиная с этой версии, нет ограничения на размер), **complex** и некоторые другие.

При присваивании атомарных объектов копируется их значение, в то время как для ссылочных копируется только указатель на объект, таким образом, обе переменные после присваивания используют одно и то же значение. Ссылочные объекты бывают изменяемые и неизменяемые. Например, строки и кортежи являются неизменяемыми, а списки, словари и многие другие объекты – изменяемыми. Кортеж в Python является, по сути, неизменяемым списком. Во многих случаях кортежи работают быстрее списков, поэтому если вы не планируете изменять последовательность, то лучше использовать кортежи.

Далее мы поговорим обо всем этом подробнее. В этой главе мы рассмотрим основные операции с переменными и поговорим детальнее о типах данных.

2.1. Имена переменных

В Python, как и в остальных языках программирования, есть переменные. Переменные в Python представлены объектами. Точнее для доступа к объекту используются переменные. При инициализации переменной (которая происходит при первом присваивании значения) в самой переменной сохраняется ссылка на объект – адрес объекта в памяти.

У каждой переменной должно быть уникальное имя, позволяющее однозначно идентифицировать объект в памяти. Имя переменной может состоять из латинских букв, цифр и знаков подчеркивания. Не смотря на то, что имена переменных могут содержать цифры, они не могут начинаться с цифры.

Также в именах переменной нужно избегать использования знака подчеркивания в качестве первого символа имени, поскольку такие имена имеют специальное значение. Имена, начинающиеся с символа подчеркивания (например, _name), не импортируются из модуля с помощью инструкции from module import *, а имена, имеющие по два символа подчеркивания (например, __name__) в начале и конце, имеют особый смысл для интерпретатора.

В качестве имени переменной нельзя использовать ключевые слова. Просмотреть список ключевых слов можно с помощью следующих инструкций (рис. 2.1):

```
>>> import keyword
>>> keyword.kwlist
```

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'awa
it', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'fina
lly', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
Ln: 6 Col: 4
```

Рис. 2.1. Ключевые слова Python

Кроме ключевых слов в качестве имени переменных не нужно использовать встроенные идентификаторы. Конечно, такие идентификаторы можно переопределить, но конечный результат будет не таким, как вы ожидаете.

Получить список встроенных идентификатор можно с помощью следующих команд:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', 'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FileExistsError',
'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'InterruptedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError',
```

```
'OSError', 'OverflowError', 'PendingDeprecationWarning',
'PermissionError', 'ProcessLookupError', 'ReferenceError',
'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'TimeoutError',
'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError',
'_', '__build_class__', '__debug__', '__doc__', '__import__',
'__loader__', '__name__', '__package__', '__spec__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals',
'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

Итак, если подытожить, то можно выделить следующие правила:

- Имя (идентификатор) может начинаться с латинской буквы любого регистра, после которой можно использовать цифры. Пример правильных имен переменных: q1, result1, a, X, MyVar
- Имя переменной не может начинаться с цифры. Пример неправильных имен переменных: 1q, 1result
- Имя переменной может начинаться с символа подчеркивания, но такие имена имеют специальное значение для интерпретатора. Примеры таких имен: _resource, __resource__
- Имя переменной не может быть ключевым словом
- Лучше не переопределять встроенные идентификаторы
- Имя переменной должно быть уникальным в пределах пространства имен

Примечание. Теоретически, имя переменной может содержать символы национальных алфавитов, но лучше такие имена не использовать.

Немного забегая вперед, хочется поговорить о пространствах имен. В любой точке программы есть доступ к трем пространствам имен (*namespaces*): локальному, глобальному и встроенному. В других языках программирования они также называются областями видимости.

Сложность скриптовых (интерпретируемых) языков программирования вроде PHP и Python заключается в том, что переменную можно объявить в любой части программы. В результате вы можете не заметить, как переменные в разных пространствах имен просто перемешались, не забыть об объявлении переменной и т.д. Например, в том же языке Pascal переменные объявляются в блоке Var, который вынесен за пределы основного блока кода. Рассмотрим небольшой пример программы на Pascal:

```
program VarTest;
var
  a, b : real;

function sum(a : real; b : real): real;
var
  res : real;
begin
  res := a + b;
  sum := res;
end;

begin
  a := 2; b := 2;
  writeln(sum(a, b);
end.
```

Четно видно, что переменные **a** и **b** являются глобальными, поскольку определены за пределами какой-либо функции, а переменная **res** является локальной, поскольку она объявлена в функции **sum**. В принципе, без нее можно было бы обойтись, но нужен был пример локальной переменной, а усложнять код не хотелось.

В Python также есть понятие глобальной и локальной переменной. Локальными считаются переменные, объявленные внутри функции (о функциях мы поговорим позднее). Простота Pascal (и многих других переменных) в

том, что переменные объявляются в одном блоке (var), при этом жестко определен тип переменной.

В Python (как и в PHP) все иначе. Какого-либо оператора или блока объявления переменной просто не существует. Объявлением переменной считается присваивание ей значения. Вот у вас может быть программа на 2000 строк и переменная res, хранящая результат, может встречаться лишь в предпоследней строке. И это будет правильно с точки зрения Python.

С типом переменной тоже не все так просто. Если в С типы определяются жестко – при объявлении переменной, то в Python типы плавающие:

```
>>> x = 1
>>> x = "test"
>>> print(x)
test
>>>
```

Сначала переменная **x** была у нас целым числом. Затем она превратилась в строку со значением "test". Попробуйте вы проделать такое в Pascal или С – вы получите ошибку несоответствия типа – ну нельзя переменной, которая была изначально запланирована для хранения числа, присвоить строку. В Python такое возможно, что тоже не добавляет ясности вашим программам. Поэтому при использовании переменных в Python нужно быть очень внимательным.

Я рекомендую, особенно начинающим программистам, объявлять все необходимые переменные в начале вашей программы путем присваивания им начальных значений. Если начального значения нет, используйте 0 для числа или "" для строки. Также комментируйте назначение переменных, если по их имени нельзя однозначно сказать, для чего они предназначены. Так вам будет гораздо проще и вы привыкните к некоторой дисциплине.

Также рассмотрим некоторые рекомендации, позволяющие навести порядок в вашем коде и сделать его более удобным для чтения и поддержки в будущем:

1. Хотя переменную можно объявить в любом месте программы, но до первого использования, рекомендуется объявлять переменные в начале программы. Там же можно произвести инициализацию переменных. Об этом мы только что говорили.

2. Неплохо бы снабдить переменные комментариями, чтобы в будущем не забыть, для чего используется та или иная переменная.

3. Имя переменной должно описывать ее суть. Например, о чем нам говорит переменная **s**? Это может быть все, что угодно и сумма (summ), и счет (score) и просто переменная-счетчик, когда вы вместо **i** почему-то используете **s**. Когда же вы называете переменную score, сразу становится понятно, для чего она будет использоваться.

4. Придерживайтесь одной и той же схемы именования переменных. Например, если вы уже назвали переменную high_score (нижний регистр и знак подчеркивания), то переменную, содержащую текущий счет пользователя называйте user_score, но никак не userScore. С синтаксической точки зрения никакой ошибки здесь нет, но код будет красивее, когда будет использоваться одна схема именования переменных.

5. Традиции языка Python рекомендую начинать имя переменной со строчной буквы и не использовать знак подчеркивания в качестве первого символа в имени переменной. Все остальное – считается дурным тоном.

6. Не создавайте слишком длинные имена переменных. В таких именах очень легко допустить опечатку, что не очень хорошо. Да и длинные имена переменных сложно "тянуть" за собой. Если нужно использовать в одной строке несколько переменных, то длинные названия будут выходить за ширину экрана. Максимальная рекомендуемая длина имени переменной – 15 символов.

2.2. Типы данных

В Python есть типы данных. При присвоении переменной значения тип данных выбирается автоматически, согласно присваиваемому значению. Тип данных может меняться на протяжении протяжение программы несколько раз – столько раз, сколько ей присваивают значения разных типов. Поддерживаемые типы данных приведены в таблице 2.1.

Таблица 2.1. Типы данных в Python

Тип данных	Описание
bool	Логический тип данных. Может содержать только два значения – *true* (истина) или *false* (ложь), что соответствует числам 1 и 0

bytearray	Изменяемая последовательность байтов
bytes	Неизменяемая последовательность байтов
complex	Комплексные числа
dict	Словарь. Похож на ассоциативный массив в PHP
ellipsis	Используется для получения среза. Определяется или ключевым словом Ellipsis или тремя точками
float	Вещественные числа
frozenset	Неизменяемое множество
function	Функция
int	Целые числа. Размер числа ограничен только размером доступной оперативной памяти
list	Список. Аналогичен массивам в других языках программирования
module	Модуль
NoneType	Пустой объект, объект без значения (точнее со значением *None*, что в других языках соответствует *null*)
set	Множество (набор уникальных объектов)
str	Unicode-строка
tuple	Кортеж

type	Типы и классы данных

Узнать тип данных можно с помощью функции type():

```
>>> type(x)
<class 'int'>
>>> x = "abc"
>>> type(x)
<class 'str'>
```

Все типы данных в Python можно разделить на *неизменяемые и изменяемые*. К неизменяемым типам данных относятся числа, строки, кортежи и bytes. К изменяемым относятся списки, словари и bytearray.

Также можно говорить о *последовательностях и отображениях*. К последовательностям относятся строки, списки, кортежи, типы **bytes** и **bytearray**. К отображениям – словари.

Последовательности и отображения поддерживают механизмы итераторов, который позволяет произвести обход всех элементов с помощью метода __next__() или функции **next**. Пример:

```
>>> m = [1, 2, 3]
>>> i = iter(m)
>>> i.__next__()
1
>>> next(i)
2
>>> next(i)
3
>>>
```

Использование метода __next__() и функции ***next()*** на практике наблюдается редко. Чаще всего используется цикл **for in**:

```
>>> for i in m:
  print(i)

1
2
3
>>>
```

Списки, кортежи, множества и словари будут рассмотрены в следующих главах, а пока рассмотрим, как в Python осуществляется присваивание переменной значения.

2.3. Присваивание значений

Для присваивания значения используется оператор =. Переменной, как и в другом языке программирования, вы можете присвоить:

- Обычное значение (константу):

```
x = 1   # Переменной x присвоено значение 1 (число)
FirstName = "Denis" # Переменной присвоена
строковая константа "Denis"
```

- Значение другой переменной

```
a = x
```

- Результат вычисления выражения

```
y = x * a + x
```

- Результат вычисления функции

```
res = func(y)
```

Как уже отмечалось, в Python используется динамическая типизация, то есть тип данных переменной изменяется в зависимости от присваиваемого ей значения. После присваивания значения в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел, строк и кортежей, но этого нельзя делать для изменяемых объектов. Рассмотрим небольшой пример. Судя по следующему коду, мы создали два разных объекта:

```
>>> a = b = [5, 4, 3]
>>> a, b
([5, 4, 3], [5, 4, 3])
```

А теперь попробуем изменить объект **a**:

```
>>> a[1] = 6
```

```
>>> a, b
([5, 6, 3], [5, 6, 3])
```

Как видите, в переменной хранится только ссылка на объект, а не сам объект. Поэтому в переменных **a** и **b** содержится ссылка на один и тот же объект. Следовательно, изменение одной переменной приводит к изменению значения и другой переменной, точнее к изменению объекта, на которого ссылается вторая переменная.

С числами, которые являются неизменяемыми объектами, вполне можно использовать групповое присваивание и получить ожидаемый результат:

```
>>> a = b = 1
>>> a = 2
>>> a, b
(2, 1)
```

Проверить, ссылаются ли переменные на один и тот же объект, можно с помощью оператора **is**. Например:

```
>>> a = b = [5, 4, 3]
>>> a is b
True
>>> b is a
True
```

Как видите, оператор **is** вернул значение *True*, что означает, что переменные **a** и **b** ссылаются на один и тот же объект в памяти. А теперь не будем использовать групповое присваивание, но присвоим переменным **a** и **b** одно и то же значение:

```
>>> a = [5, 4, 3]
>>> b = [5, 4, 3]
>>> a is b
False
>>> b is a
False
```

Теперь переменные **a** и **b** ссылаются на разные объекты в оперативной памяти. Просмотреть, сколько ссылок есть на тот или иной объект, можно с помощью метода sys.getrefcount():

```
>>> a = 5; b = 5; c = 5;
>>> sys.getrefcount(5)
105
```

Когда число ссылок на объект станет равно 0, объект будет удален из памяти.

Кроме группового присваивания в Python поддерживается позиционное присваивание, когда нужно присвоить разные значения сразу нескольким переменным, например:

```
>>> a, b, c = 5, 4, 3
>>> a, b, c
(5, 4, 3)
```

По обе стороны оператора = можно указать последовательности (строки, списки, кортежи, bytes и bytearray), но такие сложные операторы присваивания встречаются в "природе" довольно редко и я бы рекомендовал избегать их использования, если вы хотите сделать программу понятной и читаемой:

```
>>> a, b, c = "abc"
>>> x, y, z
( 'a', 'b', 'c')
>>> a, b, c = [1, 2, 3]
>>> a, b, c
(1, 2, 3)
>>> [a, b, c] = (1, 2, 3)
>>> a, b, c
(1, 2, 3)
```

Количество элементов слева и справа должно совпадать, иначе вы получите сообщение об ошибке:

```
>>> a, b, c = 1, 2
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    a, b, c = 1, 2
ValueError: need more than 2 values to unpack
>>>
```

Если справа от оператора = указано больше значений, чем переменных слева, все лишние элементы могут быть помещены в последнюю переменную. Для этого перед этой переменной нужно указать звездочку (*):

```
>>> a, b, *c = 1, 2, 3, 4
```

```
>>> a, b, c
(1, 2, [3, 4])
>>>
```

Однако такая возможность появилась в Python 3 и в Python 2.7 она не поддерживается.

> ***Примечание.*** Звездочку можно указать только перед одной переменной, иначе получите следующую ошибку:
>
> *SyntaxError: two starred expressions in assignment*

2.4. Проверка типа данных и приведение типов

Как уже отмечалось ранее, функция *type()* позволяет определить тип переменной. Например:

```
>>> a = "1"
>>> type(a)
<class 'str'>
>>>
```

Данную функцию можно использовать не только для вывода типа, но и для сравнения возвращаемого нею значения с названием типа данных:

```
>>> if type(a) == str:
  print("String");

String
```

Иногда нужно преобразовать один тип данных в другой. Эта операция называется приведением типа. Стоит отметить, что далеко не всегда можно преобразовать один тип данных в другой без потери самих данных. Некоторые типы данных вообще не совместимы. Например, никак нельзя преобразовать вещественное число в целое без потери данных. А при преобразовании строки в число вы вообще увидите сообщение об ошибке:

```
>>> int("String")
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    int("String")
ValueError: invalid literal for int() with base 10: 'String'
```

В таблице 2.2 перечислены функции приведения типов, а также примеры их использования.

Таблица 2.2. Функции приведения типов

Функция	Описание	Пример
bool()	Преобразование объекта в логический тип данных	`>>> bool(1)` `True`
int()	Преобразование объекта в целое число. Обратите внимание: дробная часть потеряна	`>>> int(5.5)` `5`
float()	Преобразование целого числа в вещественное	`>>> float(5)` `5.0`
str()	Преобразование объекта в строку	`>>> str([5, 4, 3])` `'[5, 4, 3]'`
bytes()	Преобразует строку в объект типа bytes(). Первый параметр – это строка, второй – кодировка, третий параметр необязательный и может указывать способ обработки ошибок (strict, replace, ignore)	`>>> bytes("String",` `"utf-8")` `b'String'`
bytear-ray()	Преобразует строку в объект типа bytearray	`>>>` `bytearray("Hello",` `"utf-8")` `bytearray(b'Hello')`
list()	Используется для преобразования последовательности в список	`>>> list("Hello")` `['H', 'e', 'l',` `'l', 'o']`

tuple()	Преобразует последовательность в кортеж	`>>> tuple("Hello")` `('H', 'e', 'l', 'l', 'o')`

Зачем необходимо преобразование типов, если в Python используется динамическая типизация и типы приводятся автоматически? Далеко не всегда функции возвращают значения в ожидаемом типе. Представим, что нам нужно написать простейшую программу, вычисляющую сумму двух чисел, введенных пользователем:

```
x = input("Введите x = \n")
y = input("Введите y = \n")
print(x + y)
input()
```

Запустите программу и введите числа 5 и 7. Хотя вы ввели числа, функция input() всегда возвращает введенное значение, как строку. В результате вместо числа 12 вы увидели строку 57 (рис. 2.2).

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> x = input("Введите x = \n")
Введите x =
5
>>> y = input("Введите y = \n")
Введите y =
7
>>> print(x + y)
57
>>>
Ln: 11 Col: 4
```

Рис. 2.2. Неожиданный результат работы программы

Изменим программу так:

```
x = float(input("Введите x = \n"))
y = float(input("Введите y = \n"))
```

```
print(x + y)
input()
```

Здесь мы приводим введенное пользователем значение к типу **float**, а затем вычисляем сумму двух вещественных чисел. В результате получаем значение 15.0, что соответствует нашим ожиданиям (рис. 2.3).

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Type "help", "copyright", "credits" or "license()" for more information.
>>> x = input("Введите x = \n")
Введите x =
5
>>> y = input("Введите y = \n")
Введите y =
7
>>> print(x + y)
57
>>> x = float(input("Введите x = \n"))
Введите x =
5
>>> y = float(input("Введите y = \n"))
Введите y =
7
>>> print(x + y)
12.0
>>> |
Ln: 19 Col: 4
```

Рис. 2.3. Теперь программа работает, как нужно

2.5. Удаление переменной

Для удаления переменной используется инструкция **del**:

```
del <переменная1>[, …, <переменнаяN>]
```

Пример:

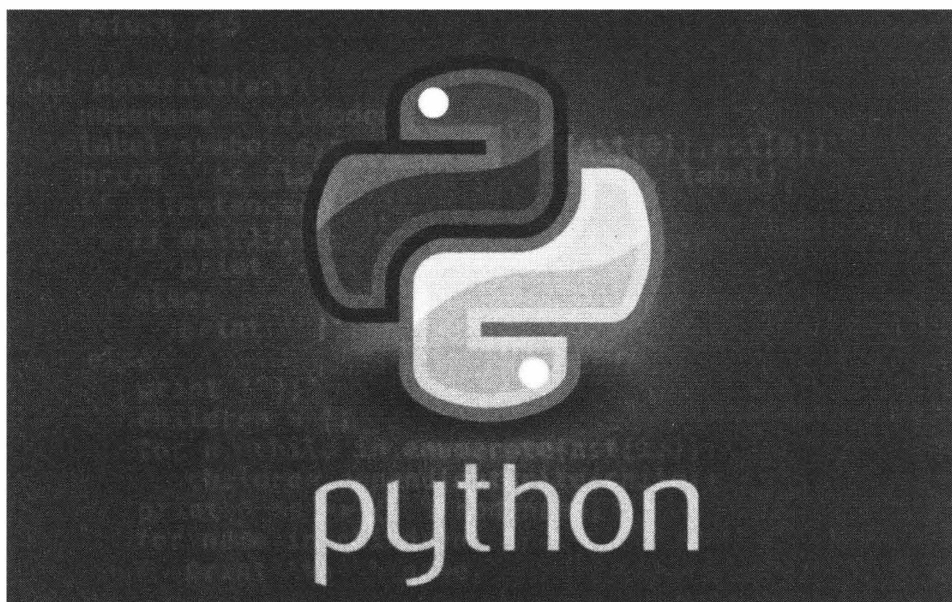
```
>>> z = 1
>>> print(z)
1
>>> del z
>>> print(z)
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    print(z)
NameError: name 'z' is not defined
>>>
```

Чтобы удалить несколько переменных, просто перечислите их в инструкции **del** через запятую:

```
del a, b, c
```

ГЛАВА 3.

ОПЕРАТОРЫ



Операторы производят определенные действия с данными. Например, математические операторы выполняют арифметические вычисления, двоичные операторы – производят манипуляции с отдельными битами данных и т.д.

3.1. Математические операторы и работа с числами

3.1.1. МАТЕМАТИЧЕСКИЕ ОПЕРАТОРЫ

Числа – немаловажный элемент любой компьютерной программы. Можно сказать, что ни одна мало-мальски полезная программа не обходится без применения чисел.

Как и в любом другом языке программирования (ну, почти в любом), в Python имеются следующие базовые математические операторы (табл. 3.1).

Таблица 3.1. Математические операторы в Python

Оператор	Действие
+	Сложение
-	Вычитание
*	Умножение
/	Обычное деление
//	Деление с остатком
%	Остаток от деления
**	Возведение в степень

Результатом оператора / всегда является вещественное число, даже если вы делите два целых числа и нет остатка. Вам кажется, что так и должно быть? В принципе да, в Python 3 так и есть. А вот в Python 2 при делении двух целых чисел возвращалось целое число, а остаток просто отбрасывался. В Python 3 оператор деления работает, как обычно.

Теперь рассмотрим примеры использования математических операторов. Обязательно обратите внимание на используемый тип данных операндов и тип данных возвращаемого результата:

```
>>> 2 + 2       # Два целых числа, результат - целое число
4
>>> 2.5 + 2     # Одно целое, одно вещественное, результат -
вещественное
4.5
>>> 2.5 + 2.5  # Два вещественных, результат - вещественное
5.0
>>> 100 - 20
80
>>> 100.5 - 80.5
20.0
>>> 5 * 5
25
>>> 5.25 * 5.25
27.5625
>>> 5 * 2.5
12.5
```

```
# Обратите внимание на разницу между операторами / и //

>>> 100 / 20
5.0
>>> 100 / 33
3.0303030303030303
>>> 100 // 5
20
>>> 100 // 33
3

>>> 100 % 33
1
>>> 2 * 2
4
>>> +100, -20, -5.0
(100, -20, -5.0)
>>>
```

При выполнении операций над вещественными числами нужно учитывать точность вычислений, иначе вы можете получить довольно неожиданные результаты. Например:

```
>>> 0.5 - 0.1 - 0.1 - 0.1
0.20000000000000004
>>> 0.5 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1
2.7755575615628914e-17
```

В первом случае результат вполне предсказуем – мы получили 0.2. А вот во втором случае мы ожидали 0, а получили значение, отличное от нуля.

Поэтому если вы разрабатываете финансовые приложения на Python, где важна точность, лучше использовать модуль Decimal:

```
>>> from decimal import Decimal
>>> Decimal("0.5") - Decimal("0.1") - Decimal("0.1") -
Decimal("0.1") - Decimal("0.1") - Decimal("0.1")
Decimal('0.0')
```

3.1.2. ПРИМЕР: ВЫЧИСЛЕНИЕ ВРЕМЕНИ В ПУТИ

Напишем небольшую программу, вычисляющую время автомобиля в пути. Пользователь должен будет ввести расстояние, которое нужно проехать, а также планируемую среднюю скорость автомобиля.

Листинг 3.1. Вычисление времени в пути

```
dist = 0       # Расстояние, которое нужно проехать
speed = 0      # Средняя скорость авто, км /ч

dist = int(input("Расстояние: "))
speed = int(input("Планируемая средняя скорость: "))

time = dist * 60 / speed

print("Время в пути ", time, " минут.")
```

Посмотрим, что есть в нашей программе. Первым делом мы инициализируем две переменные – **dist** и **speed**. Python не требует обязательной инициализации переменной. Мы сделали это так, чтобы добавить комментарий и знать, для чего используется та или иная переменная.

Далее мы получаем расстояние и среднюю скорость:

```
dist = int(input("Расстояние: "))
speed = int(input("Планируемая средняя скорость: "))
```

Обратите внимание: мы используем преобразование типа и явно указываем, что прочитанное значение должно быть типа **int**.

Затем мы вычисляем время движения автомобиля по формуле:

```
time = dist * 60 / speed
```

60 здесь – количество минут в одном часе. После того, как время вычислено, мы его выводим.

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========================= RESTART: C:\temp\py\time.py =========================
Расстояние: 120
Планируемая средняя скорость: 80
Время в пути  90.0  минут.
>>>
```

Рис. 3.1. Результат работы программы

3.1.3. ПРИМЕР: ВЫЧИСЛЕНИЕ РАСХОДА ТОПЛИВА

Данный пример демонстрирует работу с дробными числами. Ранее мы вычисляли время в пути и вводили два целых параметра. Теперь мы будем также вводить два параметра, но они с большей долей вероятности могут быть дробными.

Листинг 3.2. Вычисления расхода топлива

```
consum = 0    # Средний расход 10.5 л/100 км
dist = 0     # Расстояние, км

consum = float(input("Средний расход топлива л/100 км: "))
dist = float(input("Расстояние, км:"))

result = consum * dist / 100

print("Необходимо ", result, " л.")
```

Принцип программы такой же, как в предыдущем случае, но мы хотим получить дробные значения, поэтому мы используем функцию *float()*, которая приводит строковое значение к дробному.

> ***Внимание!*** Обратите внимание, что в качестве разделителя целой и дробной части используется точка, а не запятая! То есть, если вы введете 10.5, программа будет работать, а если вы введете 10,5, то получите сообщение об ошибке:

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
======================== RESTART: C:\temp\py\bc1.py ========================
Средний расход топлива л/100 км: 11
Расстояние, км:660
Необходимо  72.6  л.
>>>
======================== RESTART: C:\temp\py\bc1.py ========================
Средний расход топлива л/100 км: 11
Расстояние, км:80
Необходимо  8.8  л.
>>>
======================== RESTART: C:\temp\py\bc1.py ========================
Средний расход топлива л/100 км: 18
Расстояние, км:100
Необходимо  18.0  л.
>>>
Ln: 23 Col: 4
```

Рис. 3.2. Программа в действии

```
Traceback (most recent call last):
  File "E:/Python39/samples/3-2.py", line 4, in <module>
    consum = float(input("Средний расход топлива л/100 км: "))
ValueError: could not convert string to float: '10,5'
```

Данное сообщение говорит о том, что невозможно конвертировать строковое значение "10,5" в float-значение.

3.1.4. ВЫБОР ПРАВИЛЬНОГО ТИПА ДАННЫХ

В предыдущих примерах мы явно применяли *int()* и *float()* для приведения прочитанного с ввода значения к числовому типу. А что будет, если не выполнять приведения типа? Давайте посмотрим. Напишем программу, подсчитывающую стоимость содержания автомобиля.

Листинг 3.3. Стоимость содержания автомобиля

```
service = input("Стоимость ТО: ")
fuel = input("Стоимость топлива: ")
tax = input("Транспортный налог: ")
tuning = input("Тюнинг и прочие доработки: ")
insurance = input("ОСАГО: ")

total = service + fuel + tax + tuning + insurance

print("Всего: ", total)
```

Вывод изображен на рис. 3.3. Явно не то, что мы хотели. По умолчанию все введенные значения считаются строковыми, и интерпретатор просто склеил строки в одну большую строку.

```
========================= RESTART: C:\temp\py\cost.py =========================
Стоимость ТО: 15000
Стоимость топлива: 154000
Транспортный налог: 32000
Тюнинг и прочие доработки: 50000
ОСАГО: 6000
Всего:  15000154000320005000006000
>>>
```

Рис. 3.3. Стоимость содержания автомобиля. Ошибка!

Именно поэтому нам нужно явно указывать тип прочитанного значения. Исправим ошибку (рис. 3.4).

```
service = float(input("Стоимость ТО: "))
fuel = float(input("Стоимость топлива: "))
tax = float(input("Транспортный налог: "))
tuning = float(input("Тюнинг и прочие доработки: "))
insurance = float(input("ОСАГО: "))

total = service + fuel + tax + tuning + insurance

print("Всего: ", total)
```

Теперь, думаю, вы понимаете, зачем мы использовали *int()* и *float()* в предыдущих примерах.

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Стоимость ТО: 15000
Стоимость топлива: 154000
Транспортный налог: 32000
Тюнинг и прочие доработки: 50000
ОСАГО: 6000
Всего:  15000154000032000500006000
>>>
======================== RESTART: C:\temp\py\cost.py ========================
Стоимость ТО: 15000
Стоимость топлива: 154000
Транспортный налог: 32000
Тюнинг и прочие доработки: 50000
ОСАГО: 6000
Всего:  257000.0
>>> |
Ln: 39 Col: 4
```

Рис. 3.4. Стоимость содержания автомобиля. Правильная версия

3.2. Операторы для работы с последовательностями

Операторы для работы с последовательностями используют в качестве своих операндов последовательности – строки, списки, кортежи. К этим операторам относят следующие:

- **+** — конкатенация;

- ***** — повторение;
- **in** — проверка на вхождение.

Примеры использования операторов:

```
>>> "Hello, " + "world!"        # Строки
'Hello, world!'
>>> [1, 2, 3] + [ 4, 5, 6]      # Списки
[1, 2, 3, 4, 5, 6]
>>> (1, 2) + (3, 4)             # Кортежи
(1, 2, 3, 4)
```

Оператор **+** объединяет две последовательности.

```
>>> "a" * 5
'aaaaa'
>>> [1] * 5
[1, 1, 1, 1, 1]
>>> (1, 2) * 3
(1, 2, 1, 2, 1, 2)
```

Оператор ***** создает новую последовательность. В качестве исходной последовательности используется последовательность, заданная слева, а операнд справа задает количество повторов указанной последовательности.

```
>>> "s" in "String"
False
>>> "s" in "string"
True
>>> 3 in [1, 2, 3]
True
>>> 4 in (5, 5, 5)
False
```

Как видите, оператор вхождения (in) возвращает True, если операнд слева входит в состав последовательности, указанной операндом справа. В противном случае оператор возвращает False.

3.3. Операторы присваивания

Операторы этой группы используются для сохранения значения в переменной:

- = — присваивает переменной значение

- += — увеличивает значение переменной на указанную величину (или производит конкатенацию – для строк)
- -= — уменьшает значение переменной на указанную величину
- *= — умножает значение переменной на указанную величину (для строк этот оператор означает повтор)
- /= — делит значение переменной на указанную величину
- //= — то же, что и /=, но деление происходит с округлением вниз и присваиванием
- %= — деление по модулю и присваивание
- **= — возведение в степень и присваивание

Примеры (следите за возвращаемым значением):

```
>>> a = 10; a
10
>>> a += 5; a
15
>>> s = "Hel"; s += "lo"; s # Для строк конкатенация
'Hello'
>>> a -= 5; a
10
>>> a *= 2; a
20
>>> s *= 2; s                # Для строк - повтор
'HelloHello'
>>> a /= 2; a
10.0
>>> a //= 3; a
3.0
>>> a %= 2; a
1.0
>>> a **= 5; a          # Возведение в степень 1 ^ 5 = 1
1.0
```

3.4. Двоичные операторы

В современном мире вы будете редко иметь дело с двоичными операторами, разве что захотите разработать какой-то свой собственный алгоритм шифрования. Двоичные операторы используются для манипуляции над отдельными битами:

- ~ — двоичная инверсия (значение бита изменяется на противоположное: 1 на 0, 0 на 1)
- & — двоичное И
- | — двоичное ИЛИ
- ^ — двоичное исключающе ИЛИ
- << — сдвиг влево (сдвигает двоичное представление числа влево на один или несколько разрядов, разряды справа заполняются нулями)
- >> — сдвиг вправо (сдвигает двоичное представление числа вправо на один или несколько разрядов, разряды слева заполняются нулями, если число положительное, а если число отрицательное – единицами)

3.5. Приоритет выполнения операторов

Последовательность вычисления выражений зависит от приоритета выполнения операторов. Все мы знаем, что сначала выполняются умножение и деление, а потом уже сложение и вычитание, поэтому результат следующего выражения будет 6, а не 8:

```
a = 2 + 2 * 2
```

Это основы. Но в Python операторов гораздо больше, чем в математике, поэтому нужно учитывать приоритет каждого оператора. Далее приведены операторы в порядке убывания приоритета. Операторы одного приоритета выполняются слева направо:

1. -x, +x, ~x, **
2. *, %, /, //
3. +, -
4. <<, >>
5. &
6. ^
7. |
8. =, +=, -=, *=, /=, //=, %=, **=

Если вам сложно запомнить приоритет операторов, хочется большей однозначности или нужно изменить приоритет выполнения, используйте скобки. Результатом следующего выражения будет уже 8, а не 6:

```
a = (2 + 2) * 2
```

Сначала будет вычислено значение в скобках (4), а потом уже будет произведено умножение на 2.

3.6. Простейший калькулятор

Для закрепления материала о различных операторах разработаем простейший калькулятор, то есть программу, умеющую выполнять над двумя вещественными числами арифметические операции (сложение, вычитание, умножение, деление) и завершающуюся по желанию пользователя.

Наш калькулятор будет работать так:

1. Запустить бесконечный цикл. Выход из него осуществлять с помощью оператора **break**, если пользователь вводит определенный символ вместо знака арифметической операции.
2. Если пользователь ввел знак, который не является ни знаком арифметической операции, ни символом – "прерывателем" работы программы, то вывести сообщение о некорректном вводе.
3. Если был введен один из четырех знаков операции, то запросить ввод двух чисел.
4. В зависимости от знака операции выполнить соответствующее арифметическое действие.
5. Если было выбрано деление, то необходимо проверить, не является ли нулем второе число. Если это так, то сообщить о невозможности деления.

Код программы приведен в листинге 3.4.

Листинг 3.4. Калькулятор

```
print("*" * 15, " Калькулятор ", "*" * 10)
print("Для выхода введите q в качестве знака операции")
while True:
   s = input("Знак (+,-,*,/): ")
   if s == 'q': break
```

```
if s in ('+','-','*','/'):
   x = float(input("x="))
   y = float(input("y="))
   if s == '+':
      print("%.2f" % (x+y))
   elif s == '-':
      print("%.2f" % (x-y))
   elif s == '*':
      print("%.2f" % (x*y))
   elif s == '/':
      if y != 0:
         print("%.2f" % (x/y))
      else:
         print("Деление на ноль!")
else:
   print("Неверный знак операции!")
```

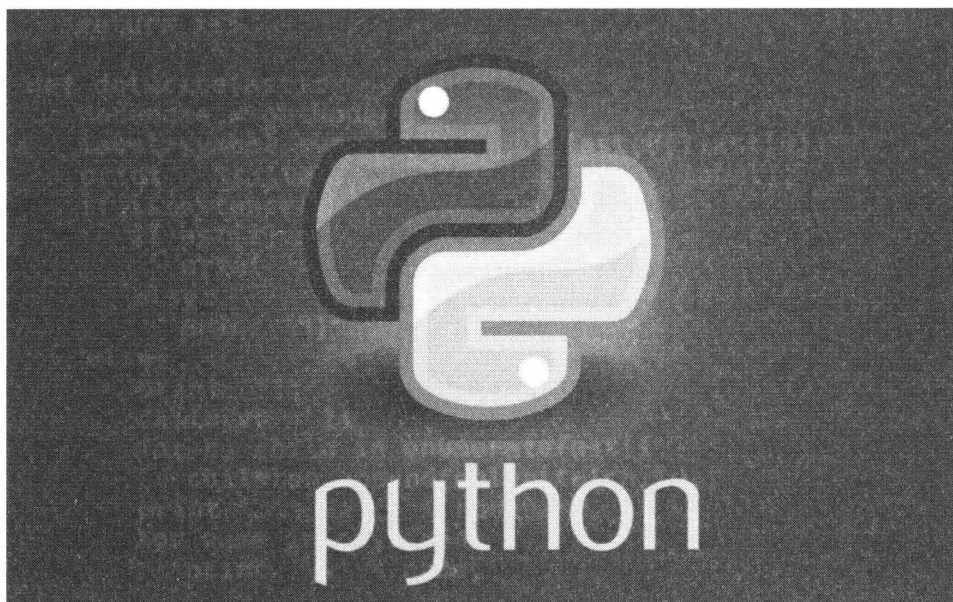
Посмотрим на вывод программы. Обратите внимание, как она реагирует на неверный ввод, например, если введено число вместо знака операции, или был введен 0 вместо **y** при делении:

```
***************  Калькулятор  **********
Для выхода введите q в качестве знака операции
Знак (+,-,*,/): +
x=12
y=13
25.00
Знак (+,-,*,/): -
x=100
y=25
75.00
Знак (+,-,*,/): /
x=9
y=3
3.00
Знак (+,-,*,/): /
x=9
y=0
Деление на ноль!
Знак (+,-,*,/): \
Неверный знак операции!
Знак (+,-,*,/): *
x=1.25
y=4
```

```
5.00
Знак (+,-,*,/): q
>>>
```

ГЛАВА 4.

ЦИКЛЫ И УСЛОВНЫЕ ОПЕРАТОРЫ



4.1. Условные операторы

4.1.1. ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ

В любой программе (если не считать самых простых) встречаются условные операторы. Данные операторы позволяют выполнить отдельный участок программы (или наоборот, не выполнить) в зависимости от значения логического выражения. Логические выражения могут вернуть только два значения: *True* (истина) или *False* (ложь), которые ведут себя как числа 1 и 0 соответственно.

Логическое значение можно хранить в переменной:

```
>>> a = True; b = False;
>>> a, b
(True, False)
```

Логическим значением *True* может интерпретироваться любой объект, не равный 0, не пустой. Числа, равные 0 или пустые объекты, интерпретируются как *False*.

4.1.2. ОПЕРАТОРЫ СРАВНЕНИЯ

В логических выражениях Python используются следующие операторы сравнения:

- **==** — равно;
- **!=** — не равно;
- **<** — меньше;
- **>** — больше;
- **<=** — меньше или равно;
- **>=** — больше или равно;
- **in** — проверяет вхождение элемента в последовательность, возвращает *True*, если элемент встречается в последовательности;
- **is** — проверяет, ссылаются ли две переменные на один и тот же объект. Если переменные ссылаются на один и тот же объект в памяти, оператор возвращает *True*.

> ***Внимание!*** Условные операторы в Python могут сравнивать не только числа, но и строки, например audi < bmw, поскольку audi находится по алфавиту раньше, чем bmw. Но не все в Python можно сравнить. Объекты разных типов, для которых не определено отношение порядка, нельзя сравнить с помощью операторов <, <=, >, >=. Например, вы не можете сравнить число и строку. Если вы попытаетесь это сделать, получите огромное сообщение об ошибке.

Примеры (обратите внимание на возвращаемые значения *True* и *False*):

```
>>> 5 == 5
True
>>> 5 != 6
True
>>> 5 == 6
False
>>> 100 > 99
True
>>> 100 < 99
False
>>> 100 <= 100
True
```

```
>>> 100 >= 101
False
>>> 2 in [1, 2, 3]
True
>>> a = b = 100
>>> a is b
True
>>>
```

Значение логического выражения можно инвертировать с помощью оператора **not**:

```
>>> a = b = 100
>>> not (a == 100)
False
```

Если нужно инвертировать значение оператора **in**, оператор **not** нужно указывать непосредственно перед **in** – без скобок:

```
>>> 2 not in [1, 5, 7]
True
```

При необходимости инвертирования оператора **is**, оператор **not** указывается после этого оператора:

```
>>> a is not b
False
```

При необходимости, можно указывать несколько условий сразу:

```
>>> 2 < 5 < 6
True
```

С помощью операторов **and** (И) и **or** (ИЛИ) можно объединить несколько логических выражений:

```
x and y
x or y
```

В первом случае, если x = False, то будет возвращен **x**, в противном случае – **y**:

```
>>> 2 < 5 and 2 < 6
True
>>> 2 < 5 and 6 < 2
False
```

Во втором случае если **x** = False, то возвращается **y**, в противном случае – **x**:

```
>>> 2 < 5 or 2 < 6
True
>>> 2 < 5 or 6 < 2
True
>>> 2 < 1 or 6 < 2
False
```

Далее перечислены операторы сравнения в порядке убывания приоритета:

1. <, >, <=, >=, ==, ! =, <>, is, is not, in, not in
2. **not** — логическое отрицание
3. **and** — логическое И
4. **or** — логическое ИЛИ

4.1.3. ОПЕРАТОР IF..ELSE

Оператор *if..else* называется оператором ветвления. Он, в зависимости от значения логического выражения, может выполнить или, наоборот, не выполнить какой-то участок программы. Формат этого оператора следующий:

```
if <логическое выражение>:
  <операторы, которые будут выполнены, если условие истинно>
[elif <логическое выражение>:
  <операторы, которые будут выполнены, если условие истинно>
]
[else:
  <операторы, которые будут выполнены, если условие истинно>
]
```

Напомню, что блоки в составной конструкции выделяются одинаковым количеством пробелов. Конец блока – инструкция, перед которой расположено меньшее число пробелов.

Рассмотрим небольшой пример. Сейчас мы напишем программу, которая будет запрашивать число N у пользователя. Далее, программа проверяет

введенное значение – оно больше или меньше ста – и выводит соответствующее сообщение (листинг 4.1).

Листинг 4.1. Пример использования оператора if..else

```
n = int(input("Введите N: "));
if n < 100:
  print("n < 100")
else:
  print("n > 100")
```

```
======================== RESTART: C:\temp\py\if.py ========================
Введите N: 5
n < 100
>>>
======================== RESTART: C:\temp\py\if.py ========================
Введите N: 567
n > 100
>>> |
```

Рис. 4.1. Результат выполнения листинга 4.1

У нас очень простая программа, в которой каждый блок состоит из одной инструкции, поэтому ее можно переписать так, как показано в листинге 4.2.

Листинг 4.2. Пример использования оператора if..else - 2

```
n = int(input("Введите N: "));
if n < 100: print("n < 100")
else: print("n > 100")
```

Однако не нужно злоупотреблять этим подходом. На практике лучше использовать подход, представленный в листинге 4.1. Так ваша программа будет более читабельной.

Оператор *if .. else* позволяет указывать несколько условий с помощью блоков **elif**. Пример использования такого условного оператора приведен в листинге 4.3.

Листинг 4.3. Проверка нескольких условий

```
print("""Выберите ваш браузер:
1 - Google Chrome
2 - Firefox
```

```
3 - MS Internet Explorer
4 - Opera
5 - Safari
6 - Другой""");

browser = int(input(""));
if browser == 1:
  print("Chrome");
elif browser == 2:
  print("Firefox");
elif browser == 3:
  print("MS IE");
elif browser == 4:
  print("Opera");
elif browser == 5:
  print("Safari");
elif browser == 6:
  print("Other");
```

```
========================= RESTART: C:\temp\py\4-4.py =========================
1
2
3
4
5
6
7
8
9
Все.
>>>
```

Рис. 4.2. Результат работы программы из листинга 4.3

Недостаток нашей программы – то, что она никак не реагирует, если пользователь введет число, отличное от 1 до 6. Исправить это можно с помощью еще одного блока **else**:

```
if browser == 1:
  print("Chrome");
elif browser == 2:
  print("Firefox");
elif browser == 3:
  print("MS IE");
elif browser == 4:
  print("Opera");
elif browser == 5:
```

```
  print("Safari");
elif browser == 6:
  print("Другой");
else:
  print("Неправильное значение")
```

Не забывайте указывать блок else, если нужна реакция на неопределенное в блоках elif значение.

Примечание. Если вы программировали на других языках, то вам наверняка знаком оператор switch..case. Смысл этого оператора в следующем: в switch задается выражение, значение которого сравнивается со значениями, заданными в блоках **case**. Если значение совпало, то выполняются операторы, указанные в этом блоке **case**. К сожалению, в Python нет такого оператора, и вам придется строить конструкции if..elif..else. Некоторые программисты предлагают использовать словари вместо switch..case, но данный подход не универсальный и подойдет далеко не всегда.

4.1.4. БЛОКИ КОДА И ОТСТУПЫ

Рассмотрим следующий условный оператор:

```
if age < 18:
  print("Извините, вы не можете использовать эту программу!")
```

Обратите внимание, что вторая строка написана с отступом. Отступ превращает наш код в блок. Блок – это одна или несколько идущих подряд строк с одинаковым отступом. Блок – единая конструкция.

Блоки используются, когда в случае выполнения условия нужно выполнить несколько операторов:

```
if age < 18:
  print("Извините, вы не можете использовать эту программу!")
  print("Как только исполнится 18, возвращайтесь!")
```

На другом языке программирования блоки кода, как правило, заключают в фигурные скобки:

```
if ($age < 18) {
  echo " Извините, вы не можете использовать эту программу!";
  echo " Как только исполнится 18, возвращайтесь!";
}
```

В других языках программирования в скобках какие-либо отступы соблюдать не нужно, операторы разделяются точкой с запятой, а написать вы можете их хоть в одну строчку, лишь бы они были в одних фигурных скобках.

В Python программисту нужно следить за отступами. Но с другой стороны это приучает его к порядку и делает код удобным для чтения.

4.2. Циклы

Если проанализировать все программы, то на втором месте, после условного оператора, будут операторы цикла. Используя цикл, вы можете повторить операторы, находящиеся в теле цикла. Количество повторов зависит от типа цикла – можно даже создать бесконечный цикл. В этом и есть некоторая опасность циклов – если не предусмотреть условие выхода из цикла, то может произойти зацикливание программы, когда тело цикла будет выполняться постоянно.

4.2.1. ЦИКЛ FOR

Цикл **for** в других языка называют еще циклом со счетчиком, поскольку он позволяет повторить тело цикла (инструкции внутри цикла) определенное количество раз. В Python цикл **for** больше похож на цикл **foreach** языка PHP – он позволяет перебрать элементы последовательности.

Формат цикла **for** следующий:

```
for <элемент> in <последовательность>
  <тело цикла>
[else:
  <блок, который будет выполнен, если не использовался
оператор break>
]
```

Здесь *элемент* – это переменная, через которую будет доступен текущий элемент итерации. *Последовательность* – объект, поддерживающий механизм итерации – строка, список, кортеж, словарь и т.д. *Тело цикла* – операторы, которые будут выполняться при каждой итерации цикла.

Изюминка цикла **for** в языке Python – наличие блока **else**, который задает операторы, которые будут выполнены, если внутри цикла не использовался

оператор **break**. Данный блок не является обязательным, но вы можете его использовать в контексте, показанном в листинге 4.4.

Листинг 4.4. Пример использования блока *else* в цикле *for*

```
for i in range(1, 10):
  print(i)
else:
  print("Все.")
```

Результат выполнения этого кода приведен на рис. 4.3. Как видите, сценарий вывел числа от 1 до 9 и в конце работы цикла вывел сообщение "Все." Теперь переделаем цикл так, чтобы внутри был оператор **break**, который прерывает работу цикла (лист. 4.5). Результат изображен на рис. 4.4. Как видите, если выполнение цикла прерывается оператором **break**, то операторы из блока **else** не выполняются.

```
========================= RESTART: C:\temp\py\4-4.py ===========
1
2
3
4
5
6
7
8
9
Все.
>>>
```

Рис. 4.3. Результат выполнения кода из листинга 4.4

```
==================== RESTART: C:\temp\py\4-5.py ====================
1
2
3
4
5
6
>>>
```

Рис. 4.4. Результат выполнения кода из листинга 4.5

Листинг 4.5. Цикл с оператором *break*

```
for i in range(1, 10):
  print(i)
  if i == 6:
    break
else:
  print("Все.")
```

Блок **else** не обязателен и вы можете его не использовать.

Цикл **for** можно также использовать для перебора элементов словаря, хотя словарь не является последовательностью. В листинге 4.6 приведен пример перебора элементов словаря.

Листинг 4.6. Пример перебора элементов словаря

```
dict = {"a" : 1, "b": 2}
for key in dict.keys():
  print(key, " => ", dict[key])
```

```
========================= RESTART: C:\temp\py\4-6.py =========================
a  =>  1
b  =>  2
>>> |
```

Рис. 4.5. Перебор элементов словаря

Наверное, вы заметили, что элементы словаря выведены в произвольном порядке, а не в том, который был указан при создании объекта. Чтобы упорядочить вывод словаря, его ключ нужно отсортировать функцией sorted():

```
for key in sorted(dict.keys()):
  print(key, " => ", dict[key])
```

После этого вывод будет такой, как вы ожидали:

```
c:\Python39>python dict.py
a  =>  1
b  =>  2
```

Цикл **for** можно использовать не только для прохода по последовательности чисел. Возможен проход по любой последовательности элементов. Например, вот так можно пройтись по всем буквам:

```
for letter in word:
  print(letter)
```

4.2.2. ЦИКЛ WHILE

В языке Python кроме цикла **for** есть также и цикл **while**. На этот раз данный цикл – без сюрпризов и он работает так, как в других языках программирования, а именно выполняется до тех пор, пока логическое выражение истинно:

```
while <логическое выражение>:
  <тело цикла>
[else:
  <блок, который будет выполнен, если не использовался
оператор break>
]
```

Как и у цикла **for**, у цикла **while** есть блок *else*. Оператор **while** нужно использовать очень осторожно. Если в теле цикла не предусмотреть изменение логического выражения, то можно получить бесконечный цикл, который приведет к так называемому "зацикливанию" программы. Ниже приведено несколько примеров "вечных" циклов:

```
# Условие неизменно и всегда истинно.
while True:
  print("Привет")

# В теле цикла значение n не изменяется, следовательно,
# n всегда будет < 10 и цикл будет выполняться бесконечно
n = 0
while n < 10
  print("Привет")
```

Прервать выполнение бесконечного цикла можно с помощью комбинации клавиш Ctrl + C, после чего вы увидите такой вывод:

```
Traceback (most recent call last):
  File "<pyshell#21>", line 2, in <module>
    print("Привет")
  File "E:\Python39\lib\idlelib\PyShell.py", line 1352, in write
    return self.shell.write(s, self.tags)
KeyboardInterrupt
```

Цикл **for** более безопасен – он будет закончен тогда, когда будут перебраны все элементы последовательности. Бесконечных последовательностей не бывает, поэтому рано или поздно цикл будет закончен (если, конечно, в цикле не происходит изменения последовательности). А вот за телом цикла **while** нужно следить. Чтобы не допустить бесконечного цикла нужно или предусмотреть условие выхода из цикла или предусмотреть изменение условия. Перепишем два наших проблемных цикла так, чтобы они стали "конечными":

```
# Предусматриваем условие выхода
# Тело будет выполнено 5 раз
n = 0
while True:
  print("Привет")
  n += 1
  if n == 5: break

# В теле цикла значение n увеличивается, следовательно
# как только оно достигнет 10, цикл будет прерван
n = 0
while n < 10
  print("Привет")
  n += 1
```

4.2.3. ОПЕРАТОРЫ BREAK И CONTINUE

Как уже было показано ранее, оператор **break** досрочно прерывает цикл. Оператор **continue** прерывает текущую итерацию и осуществляет переход на следующую. Пример использования этих операторов приведен в листинге 4.7.

Листинг 4.7. Операторы break и continue

```
for n in range(1, 20):
  if n == 5:
    continue
  if n == 12:
    break
  print(n)
```

Хотя последовательность содержит числа от 1 до 19 (конечное значение не входит в возвращаемое значение), число 5 не будет выведено, поскольку

оператор **continue** выполнит переход на следующую итерацию, а выполнение всего цикла будет прервано на 12-ой итерации. В итоге мы увидим числа от 1 до 11, но без числа 5 (см. рис. 4.6).

```
======================= RESTART: C:\temp\py\4-7.py =======================
1
2
3
4
6
7
8
9
10
11
>>> |
```

Рис. 4.6. Операторы break и continue (лист. 4.7)

4.2.4. ФУНКЦИЯ RANGE()

Функция *range()* позволяет сгенерировать последовательность нужной длины. По сути, функция *range()* позволяет превратить цикл **for** в его классический вариант – цикл со счетчиком, например:

```
for x in range(1, 100):
  print(x)
```

Формат функции *range()* следующий:

```
range([начало,] конец [, шаг])
```

Если у функции один параметр, то это – **конец**. При этом в качестве параметра **начало** используется 0. То есть range(0) равносильно range(0, 100). Значение параметра **конец** не включается в создаваемую последовательность, то есть при вызове range(0, 100) в последовательности будут числа от 0 до 99.

Параметр **шаг** задает инкремент. По умолчанию используется значение 1. Шаг может быть и отрицательным, поэтому вы можете не только увеличивать значение, но и уменьшать его:

```
for x in range(200, 100, -1):
  print(x)
```

В Python 2 функция *range()* возвращала просто список чисел. В Python 3 возвращается объект, поддерживающий механизм итерации. Данный объект поддерживает методы index(<значение>) и count(<значение>). Первый возвращает индекс элемента, имеющего указанное значение. Второй возвращает количество элементов с указанным значением. Примеры:

```
>>> rng = range(1, 100)
>>> rng.index(5)
4
>>> rng.count(100)
0
```

Нумерация элементов начинается с 0, поэтому элементу с числом 5 соответствует индекс 4. А вот поскольку число 100 не входит в нашу последовательность, то количество элементов, равных 100, равно 0.

Рассмотрим еще несколько примеров использования range():

```
for i in range(10):
    print(i, end=" ")

print()

for i in range(0, 50, 5):
    print(i, end=" ")

print()

for i in range(10, 0, -1):
    print(i, end=" ")
```

Вывод будет таким:

```
0 1 2 3 4 5 6 7 8 9
0 5 10 15 20 25 30 35 40 45
10 9 8 7 6 5 4 3 2 1
```

Первый цикл **for** выводит значения от 0 до 10. Мы указываем только верхнюю границу. Если нижняя граница не указана, то подразумевается, что это 0.

Второй цикл **for** работает от 0 до 5, а увеличение счетчика происходит сразу на 5 единиц. Поэтому мы увидим числа 0, 5, 10, 15 и т.д. – кратные 5.

Третий цикл работает от 10 до 0, уменьшение счетчика происходит на единицу (-1). Поэтому числа будут выведены в обратном порядке.

Функция *range()* возвращает последовательность цифр. Если ей передать в качестве аргумента положительное число, то последовательность будет охватывать числа от 0 до переданного аргумента (включая его).

Если передать функции *range()* три аргумента, как мы это сделали во втором и третьем случаях, то они будут рассматриваться как начало, конец счета и интервал. Начало – это первый элемент нашей последовательности чисел, а конечное значение в него не попадает, поэтому мы получили набор чисел `0 5 10 15 20 25 30 35 40 45` во втором случае.

4.3. Бесконечные циклы

4.3.1. БЕСКОНЕЧНЫЙ ЦИКЛ ПО ОШИБКЕ

Особое внимание уделите изменению значения управляющей переменной. Неправильно составленное условие может привести к бесконечному циклу. Рассмотрим пример зацикливания программы:

```
k = 10
while k > 5:
    print(k)
    k = k + 1
```

Здесь цикл будет выполняться, пока **k** больше 5. Изначально **k** у нас больше 5, далее значение **k** только увеличивается, поэтому мы получим бесконечный цикл – программа будет бесконечно увеличивать значение **k** и выводить его:

```
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
```

Прервать выполнение зацикленной программы можно с помощью комбинации клавиш Ctrl + C:

```
Traceback (most recent call last):
  File "E:/Python39/loop.py", line 3, in <module>
    print(k)
  File "E:\Python39\lib\idlelib\PyShell.py", line 1344, in
write
    return self.shell.write(s, self.tags)
KeyboardInterrupt
>>>
```

Как исправить бесконечный цикл. Здесь нужно или редактировать условие или же процесс изменения значения управляющей переменной. Например, можно сделать декремент управляющей переменной:

```
k = k - 1
```

Тогда программа выведет 5 чисел и завершит свою работу:

```
10
9
8
7
6
```

Если разложить наш цикл на итерации, то получится табличка, приведенная ниже (табл. 4.1)

Таблица 4.1. Итерации цикла

Номер итерации	**Значение *k***	**Проверка условия**	**Действия**
1	10	true	`print(k)   # 10` `k = k - 1 # 9`
2	9	true	`print(k)   # 9` `k = k - 1 # 8`

3	8	true	`print(k) # 8` `k = k - 1 # 7`
4	7	true	`print(k) # 7` `k = k - 1 # 6`
5	6	true	`print(k) # 6` `k = k - 1 # 5`
6	5	false	-

Можно было бы изменить и условие, например:

```
k = 10
while k < 15:
    print(k)
    k = k + 1
```

Тогда программа выведет:

```
10
11
12
13
14
```

Типичная ошибка новичков – многие вообще забывают изменять переменную в цикле. Например:

```
# Внимание! Код содержит ошибку!
k = 1
while k <= 10:
  print(k)
```

Очевидно, программист хотел, чтобы программа отобразила числа от 1 до 10, но забыл изменить значение **k** в теле цикла. Следовательно, программа будет выполняться бесконечно.

Подытожим. Чтобы не получить бесконечный цикл, необходимо:

1. Следить за начальным значением управляющей переменной
2. Проанализировать условие выхода из цикла

3. Следить за процессом изменения значения управляющей переменной: ей должны присваиваться такие значения, которые рано или поздно приведут к выходу из цикла

4.3.2. НАМЕРЕННЫЙ БЕСКОНЕЧНЫЙ ЦИКЛ

Ради справедливости нужно отметить, что иногда бесконечные циклы создаются преднамеренно, поскольку того требуют условия задачи. Например, мы пишем какую-то программу, которая обрабатывает запросы извне, например, поступающие по сетевому сокету.

В этом случае проще написать так:

```
while True:
  блок_кода
```

Не нужно использовать пример, приведенный ранее (когда значение управляющей переменной не установлено) – так ваш код будет похож на ошибочный. А когда вы указываете while True:, то вы явно сообщаете, что хотите создать бесконечный цикл.

Как все-таки прервать цикл, например, если в теле цикла было получено сообщение прекратить работу программы? Для этого нужно использовать инструкцию **break**. Например:

```
while True:
  data = read_from_socket()
  if data == "quit":
    break
```

Вы обязательно должны предусмотреть возможность выхода из бесконечного цикла. В данном случае, если значение переменной **data** будет равно "quit", то выполнение цикла будет прервано.

Нужно обязательно предусмотреть возможность выхода из цикла, поскольку прерывание цикла по нажатию Ctrl + C, во-первых, считается дурным тоном, во-вторых, приводит к прерыванию всей программы, а не только цикла.

Как и в других языках программирования, в Python есть инструкция **continue**, позволяющая пропустить итерацию. Например:

```
k = 0
while k < 17:
    k = k + 1
```

```
    if k %5 == 0:
        continue
    print(k)
```

Программа выведет:

```
1
2
3
4
6
7
8
9
11
12
13
14
16
17
```

Как видите, в списке отсутствуют значения, кратные 5. Если остаток от деления на 5 равен 0, то мы просто переходим на следующую итерацию и пропускаем текущую. В этом коде, не смотря на его простоту, очень сложно допустить ошибку. Например, если изменять значение **k** уже после проверки на кратность оператором **if**, то можно получить бесконечный цикл:

```
# Внимание! Код содержит ошибку!
k = 0
while k < 17:
    if k %5 == 0:
        continue
    print(k)
    k = k + 1
```

Давайте посмотрим, что произойдет. Представим, что **k** уже равно 4. Поскольку k < 17, начнется выполнения тела цикла. Так как k % 5 не равно 0, инструкция **continue** не будет выполнена. Цифра 4 будет выведена на экран, после чего значение **k** будет увеличено на 1 и станет равно 5.

Далее проверяется условие: значение k < 17, поэтому начинается выполнение тела цикла. В результате k % 5 мы получаем 0 и пропускаем текущую итерацию. Но проблема в том, что значение **k** мы так и не увеличили, и оно

по-прежнему равно 5. Ситуация повторяется, и так будет происходить, пока вы не нажмете Ctrl + C.

4.4. Истинные и ложные значения

Рассмотрим вот такое условие:

```
if score:
```

Странно, ведь **score** не сравнивается ни с одним значением, как так? Сама переменная **score** выступает как условие. Если значение **score** будет равно 0, то это считается ложным значением (*false*). Любое другое значение считается истинным (*true*). С тем же успехом мы могли бы написать:

```
if score > 0:
```

Но незачем делать код сложнее! Ведь можно сделать его проще!

4.5. Практический пример. Программа Уровень доступа

Логические операторы **not**, **or** и **and** представляют логические операции

НЕ, ИЛИ и И соответственно.

Логическая бинарная операция И (and) возвращает *true*, если оба операнда истинны:

```
if money and score:
```

Здесь подразумевается, если деньги и счет отличны от 0, то условие будет истинным.

Логическая бинарная операция ИЛИ (or) возвращает *true*, если один из операндов равен *true*:

```
if money or score:
```

Если одна из переменных, содержит значение, отличное от 0, то условие будет истинным.

Логическая унарная операция отрицания NOT возвращает истину, если операнд был ложным и наоборот. Вот как можно бесконечно запрашивать ввод пароля, пока он не будет введен:

```
password = ""
while not password:
  password = input("Пароль: ")
```

Данные логические операции можно использовать для составления более сложных условий в циклах и условных операторах **if**. Рассмотрим небольшой пример (лист. 4.8). Данная программа запрашивает логин и пароль и на основании этих данных определяет уровень доступа.

Листинг 4.8. Определение уровня доступа

```
level = 0       # Уровень доступа

login = ""
while not login:
  login = input("Логин: ")

password = ""
while not password:
  password = input("Пароль: ")

if login == "root" and password == "123":
    level = 10
elif login == "mark" and password == "321":
    level = 5

if level:
    print("Привет, ", login)
    print("Ваш уровень доступа: ", level)
else:
    print("Доступ запрещен!")
```

По умолчанию уровень доступа равен 0. Если это так, то доступ закрыт. Если уровень отличается от 0, то программа выводит приветствие и сообщает уровень доступа.

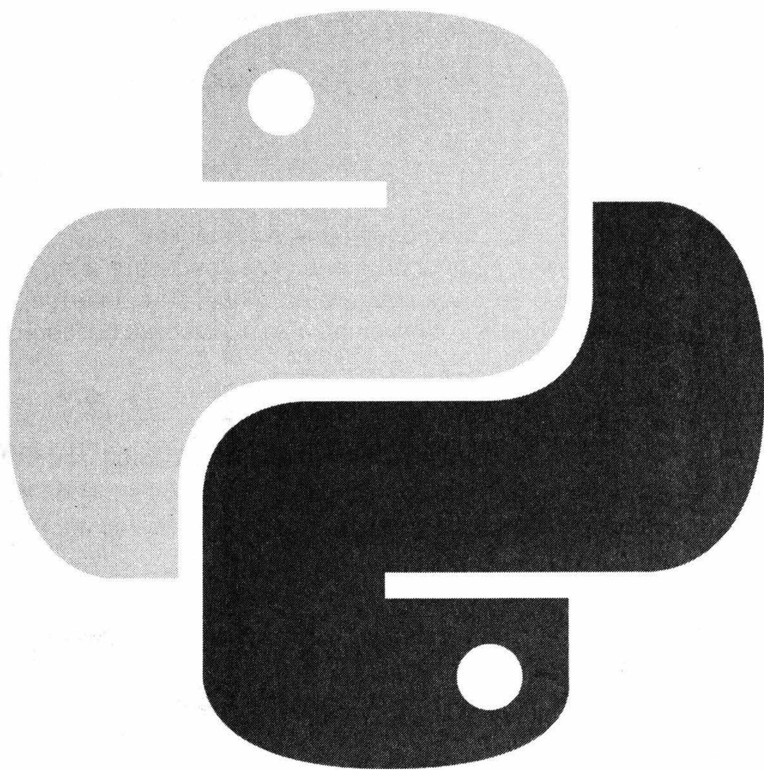
Сначала мы в двух циклах **while** запрашиваем логин и пароль. Благодаря наличию **not** мы будем запрашивать логин и пароль до тех пор, пока они не будут введены.

Далее мы сравниванием введенные значения с определенными константами и определяем уровень доступа.

Рассмотрим вывод программы:

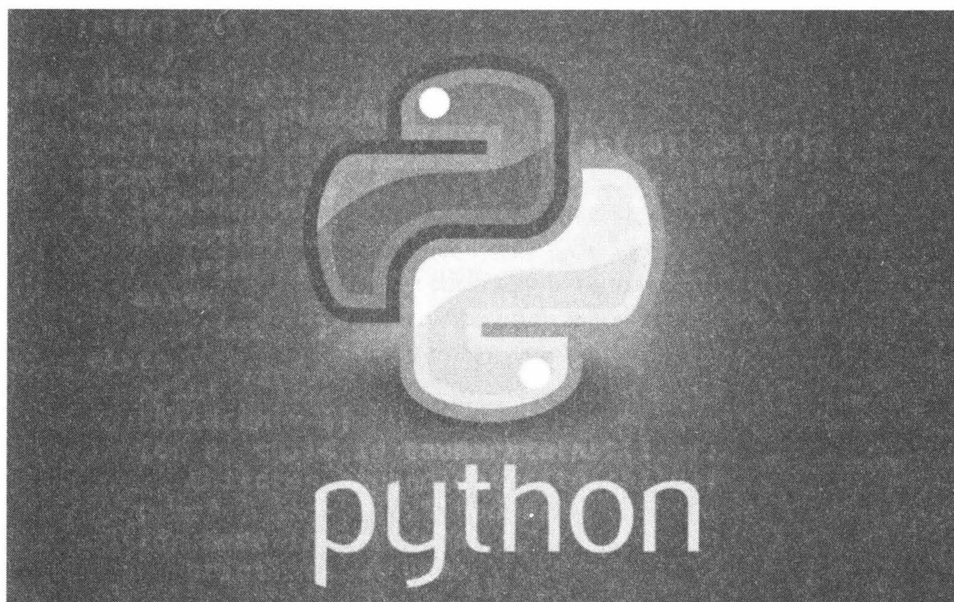
```
=========================
Логин: mark
Пароль: 321
Привет,  mark
Ваш уровень доступа:  5
>>>
=========================
Логин: root
Пароль: 1234
Доступ запрещен!
>>>
```

В первом случае были введены правильные логин и пароль. Программа сообщила уровень доступа. Во втором случае логин был введен правильно, а пароль – нет. Программа сообщила, что доступ закрыт.



ГЛАВА 5.

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ



5.1. Поддерживаемые типы чисел

Python поддерживает следующие типы чисел: **int**, **float**, **complex**. Как вы уже знаете из предыдущих глав, это целые, вещественные и комплексные числа соответственно. При операции с числами нужно помнить, что результатом операции является число более сложного типа. Например, вы хотите умножить целое число на вещественное, тогда результатом будет вещественное число.

Самым простым числовым типом является целое число. Чуть сложнее – вещественное, поскольку у него есть дробная часть. Конечно же, самым сложным типом является комплексное число.

Создать числовой объект можно так же, как и объекты остальных типов:

```
>>> a = 5; b = 2
>>> c = a * b
```

С помощью префиксов 0b (0B), 0o (0O) и 0x (или 0X) можно указать числа в двоичной, восьмеричной и шестнадцатеричной системах счисления соответственно:

```
>>> a = 0b11110000
>>> b = 0o555
>>> c = 0xfff
```

Вещественные числа могут быть представлены в экспоненциальной форме

– с точкой и буквой E, например:

```
>>> a = 5e10
>>> b = 2.5e-5
```

Вещественные числа записываются в виде:

```
Вещественная_часть+мнимая_частьJ
```

Например:

```
>>> a = 3+4J
```

Для выполнения операций повышенной точности над вещественными числами нужно использовать модуль **decimal**, например:

```
>>> from decimal import Decimal
>>> Decimal("0.2") - Decimal("0.1") - Decimal("0.1")
Decimal('0.0')
```

Модуль Decimal реализует "Общую спецификацию десятичной арифметики" IBM. Само собой, разумеется, есть огромное число параметров конфигурации, которые выходят за рамки этой книги.

Новички в Python могут использовать модуль Decimal, чтобы избежать проблем с точностью, которые имеются при работе с типом данных **float**. Однако здесь важно понять, а нужна ли вам такая точность. Тут все зависит от вашего приложения. Если вы решаете научные или технические задачи, занимаетесь компьютерной графикой или решаете большинство задач научной природы, вам будет вполне достаточно обычного типа **float**. В мире существует очень мало вещей, для которых будет недостаточно обеспечиваемой этим типом данных 17-значной точности. Таким образом, крошечные ошибки, имеющиеся в вычислениях, просто не имеют значения. К тому же производительность вычислений с типом данных **float** (в отличие от модуля Decimal) – на высоте.

Исходя из всего сказанного, основное применение модуля **decimal** – в финансовых программах. В таких программах необходимо чрезвычайно точное вычисление и даже малейшие ошибки недопустимы. Таким образом, decimal позволяет избежать таких ошибок. Также объекты decimal принято использовать при взаимодействии с базами данных, особенно при доступе к финансовым данным.

Модуль **fractions** обеспечивает поддержку рациональных чисел:

```
>>> from fractions import Fraction
>>> Fraction("0.2") - Fraction("0.1") - Fraction("0.1")
Fraction(0, 1)
```

Модуль **fractions** может быть использован для осуществления математических операций с дробями. Например:

```
>>> from fractions import Fraction
>>> a = Fraction(6, 4)
>>> b = Fraction(7, 12)
>>> print(a + b)
25/12
>>> print(a * b)
7/8

>>> # Получение числителя/знаменателя
>>> c = a * b
>>> c.numerator
7
>>> c.denominator
8

>>> # Конвертирование в float
>>> float(c)
0.875

>>> # Ограничение значения знаменателя
>>> print(c.limit_denominator(8))
7/8
>>> # Преобразование из float в дробь
>>> x = 5.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(23, 4)
>>>
```

5.2. Числовые функции

В таблице 5.1 представлены встроенные числовые функции, имеющиеся в Python.

Таблица 5.1. Встроенные числовые функции

Функция	Описание
abs(<число>)	Возвращает абсолютное значение числа: `>>> abs(-5), abs(-7.5)` `(5, 7.5)`
bin(<число>)	Преобразует **десятичное** число в двоичную систему, возвращает строку: `>>> bin(0), bin(333)` `('0b0', '0b101001101')`
divmod(a, b)	Возвращает кортеж из двух значений - (a // b, a % b)
float(<число или строка>)	Преобразует **целое** число или строку в вещественное число: `>>> float(3), float("2.2"),` `float("13.")` `(3.0, 2.2, 13.0)`
hex(<число>)	Преобразует **десятичное** число в шестнадцатеричную форму, возвращает строку
int(<объект> [, система счисления])	Преобразует объект в целое число. Второй параметр позволяет указать систему счисления: 16 – шестнадцатеричная 10 – десятичная (по умолчанию) 8 – восьмеричная 2 – двоичная Пример: `>>> int(5.5), int("50", 10),` `int("0xfff", 16), int("0o555", 8)` `(5, 50, 4095, 365)`

max(<список>) **min(<список>)**	Возвращают максимальное/минимальное значение из заданного списка. Список задается через запятую: `>>> max(4, 7, 5), min(1, 4, 9)` `(7, 1)`
oct(<число>)	Преобразует **десятичное** число в восьмеричную систему, возвращает строку
pow(<число>, <степень> [, K])	Возводит указанное число в указанную степень. Последний параметр задает остаток от деления, то есть если он указан, то возвращается остаток от деления (число возводится в степень, делится на K и возвращается остаток). Например: `>>> pow(5, 2), pow(10, 2, 2),` `pow(10, 2, 3)` `(25, 0, 1)`
round(<число> [, N])	Округляет число до ближайшего меньшего целого для чисел с дробной частью меньше 0.5 или до ближайшего большего целого для чисел с дробной частью больше 0.5. Если дробная часть равна 0.5, округление производится до ближайшего четного числа. Второй необязательный параметр N задает число знаков после точки. Пример: `>>> round(0.33), round(1.7),` `round(0.51)` `(0, 2, 1)`
sum(<последовательность> [, N])	Возвращает сумму значений элементов последовательности плюс N (N – это начальное значение). Примеры: `>>> sum([1, 2, 3]), sum([1, 2, 3],` `1)` `(6, 7)`

Встроенные функции можно использовать без указания имени модуля, в котором они находятся. Одни из самых частых операций – округление и форматирование чисел. Обе эти операции и рассмотрены далее.

5.2.1. ОКРУГЛЕНИЕ ЧИСЛОВЫХ ЗНАЧЕНИЙ

Часто нужно округлить число с плавающей запятой к числу с фиксированным числом десятичных знаков. Для простого округления можно использовать встроенную функцию round(value, ndigits). Например:

```
>>> round(1.24, 1)
1.2
>>> round(1.28, 1)
1.3
>>> round(-1.29, 1)
-1.3
>>> round(1.25371,3)
1.254
>>>
```

Функция *round()* округляет промежуточные значения к ближайшей ***четной*** цифре. То есть значения, такие как 1.5 или 2.5 будут округлены к 2. Число разрядов, передаваемых функции round() может быть отрицательным, когда округление имеет место для десятков, сотен, тысяч и т.д. Например:

```
>>> a = 2625531
>>> round(a, -1)
2625530
>>> round(a, -2)
2625500
>>> round(a, -3)
2626000
>>>
```

Не путайте округление с форматированием значения для вывода. Если

ваша цель заключается в том, чтобы просто вывести численное значение с определенным числом десятичных разрядов, вы не должны использовать *round()*. Вместо этого лучше определить желаемую точность при форматировании. Например:

```
>>> x = 1.234567
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'number - {:0.3f}'.format(x)
'number - 1.235'
>>>
```

Кроме того, не нужно округлять числа с плавающей запятой к числам с

фиксированным числом десятичных разрядов, чтобы избежать проблем точности. Пример:

```
>>> a = 3.1
>>> b = 4.2
>>> c = a + b
>>> c
7.300000000000001
>>> c = round(c, 2)
>>> c
7.3
>>>
```

Для большинства приложений, работающих с плавающей точкой, просто рекомендуется это сделать. Если для вас важно предотвращение таких ошибок (например, в финансовых приложениях), рассмотрите использование модуля decimal.

5.2.2. ФОРМАТИРОВАНИЕ ЧИСЕЛ ДЛЯ ВЫВОДА

Для форматирования одного числа для вывода, используется встроенная функция format(). Например:

```
>>> x = 9876.54321
>>> # Два десятичных места точности
>>> format(x, '0.2f')
'9876.54'
>>> # Выравнивание по правому краю, 10 символов, 1 разряд
точности
>>> format(x, '>10.1f')
'    9876.5'
>>> # Выравнивание по левому краю, 1 разряд
>>> format(x, '<10.1f')
'9876.5    '
>>> # Выравнивание по центру
```

```
>>> format(x, '^10.1f')
'  9876.5  '
>>> # Добавление разделителя тысяч
>>> format(x, ',')
'9,876.54321'
>>> format(x, '0,.1f')
'9,876.5'
>>>
```

Если вы хотите использовать экспоненциальную запись, измените **f** на **e** или **E**, в зависимости от регистра, который вы хотите использовать для экспоненциального спецификатора. Например:

```
>>> format(x, 'e')
'9.876543e+03'
>>> format(x, '0.2E')
'9.88E+03'
>>>
```

Общая форма ширины и точности в обоих случаях – '[<>^]?width[,]?(.digits)?', где **width** (ширина) и **digits** (разряды) – целые числа, а **?** показывает дополнительные части.

Форматирование значений с разделителем тысяч тоже не проблема. Однако сам разделитель зависит от настроек локали, поэтому желательно исследовать функции из модуля **locale**. Вы можете заменить символ разделителя, используя метод *translate()* строки. Например:

```
>>> separators = { ord('.'):',', ord(','):'.' }
>>> format(x, ',').translate(separators)
'9.876,54321'
>>>
```

5.3. Математические функции

Математические функции содержатся в модуле **math**, поэтому перед их использованием вам нужно импортировать этот модуль:

```
import math
```

Для работы с комплексными числами нужно импортировать модуль cmath:

```
import cmath
```

В модуле math можно найти следующие константы:

- pi — возвращает число Пи
- e – возвращает значение константы **e**

Пример:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

Математические функции приведены в таблице 5.2.

Таблица 5.2. Математические функции (модуль math)

Функция	Описание
atan2(x, y)	Аналогично atan(x/y). Если y равен 0, то возвращается pi/2
ceil(x)	Возвращает наименьшее вещественное число с нулевой дробной частью – большее, чем число **x**
exp(x)	Возвращает e**x
fabs(x)	Возвращает абсолютное значение числа **x**
floor(x)	Наибольшее вещественное число с нулевой дробной частью – меньшее, чем число **x**
fmod(x, y)	Возвращает остаток от деления x на y и эквивалентно x%y
hypot(x, y)	Возвращает длину гипотенузы прямоугольника со сторонами длиной **x** и **y** и эквивалентно sqrt(x*x+y*y)

log(x), log10(x)	Натуральный и десятичный логарифм числа **x**
modf(x)	Возвращает кортеж из пары вещественных чисел – дробной и целой части **x**
sin(x), cos(x), tan(x), asin(x), acos(x), atan(x)	Всем известные стандартные и обратные тригонометрические функции (синус, косинус, тангенс, арксинус, арккосинус, арктангенс). Значение возвращается в радианах
sinh(x), cosh(x), tanh(x)	Гиперболические синус, косинус, тангенс числа **x**
sqrt(x)	Корень квадратный числа **x**

Поскольку функции не являются встроенными, использовать их нужно так:

```
>>> import math
>>> math.log(10)
2.302585092994046
math.log10(10)
>>> math.log10(10)
1.0
```

5.4. Случайные числа. Модуль random

Модуль *random()* содержит функции для работы со случайными числами:

```
import random
```

Функции, предоставляемые этим модулем, приведены в таблице 5.3.

Таблица 5.3. Функции для работы со случайными числами

Функция	Описание
random()	Возвращает случайное вещественное число **r**, находящееся в диапазоне 0.0 < r <1.0

shuffle(<список>[, N])	N – это число от 0.0 до 1.0. Перемешивает элементы списка случайным образом. Функция работает непосредственно со списком и ничего не возвращает, поэтому будьте осторожны: лучше работать с копией списка. Если не указан второй параметр, то используется значение, возвращаемое функцией random()
choice(<последовательность>)	Возвращает случайный элемент из указанной последовательности (которая может быть представлена списком или кортежем)
uniform(a, b)	Возвращает случайное вещественное число **r**, находящееся в диапазоне a < r < b
randrange(начало, конец, шаг)	Возвращает случайное целое число **r**, находящееся в диапазоне range (начало, конец, шаг)

Примеры использования модуля **random**:

```
>>> import random
>>> random.random()
0.9922129256765113
>>> random.uniform(1,100)
64.5126755129645
>>> randrange(1,100,1)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    randrange(1,100,1)
NameError: name 'randrange' is not defined
>>> random.randrange(1,100,1)
54
```

Обратите внимание: если вызвать функцию без названия модуля, то вы получите сообщение об ошибке. Понимаю, что не очень хочется "таскать" за собой название модуля (пакета). Поэтому можете использовать конструкцию:

```
from random import *
```

После этого можно использовать функции как обычно:

```
>>> random()
0.9942452546319136
```

Аналогично, вы можете импортировать все функции из **math** и использовать их подобно встроенным функциям.

Функция *random.choice()* может использоваться для выбора случайного элемента последовательности:

```
>>> import random
>>> seq = [8, 7, 6, 5, 4, 3, 2, 1]
>>> random.choice(seq)
5
>>> random.choice(seq)
1
>>> random.choice(seq)
7
```

Случайная выборка из N элементов с использованием *random.sample()*:

```
>>> random.sample(seq, 3)
[6, 7, 5]
>>> random.sample(seq, 3)
[2, 3, 4]
```

Если вам нужно просто перемешать элементы последовательности, используйте *random.shuffle()*:

```
>>> random.shuffle(seq)
>>> seq
[5, 1, 6, 8, 3, 2, 4, 7]
```

Чтобы создать случайные целые числа, используйте *random.randint()*:

```
>>> random.randint(0,100)
47
>>> random.randint(0,100)
97
```

Модуль **random** вычисляет случайные числа, используя алгоритм Вихря Мерсенна (Mersenne Twister). Это – детерминированный алгоритм (то есть его не нужно предварительно инициализировать, как в PHP), но вы можете

настроить генератор случайных чисел на другую последовательность, используя функцию *random.seed()*:

```
random.seed()
```

5.5. Значения Infinity и NaN

В Python есть два специальных значения:

- inf — бесконечность
- NaN (Not a Nymber) — не число

У Python нет специального синтаксиса для представления этих специальных значений с плавающей запятой, но они могут быть созданы с помощью функции *float()*. Например:

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

Для проверки на присутствие таких значений используйте функции *math.isinf()* и *math.isnan()*. Например:

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

5.6. Вычисления с большими числовыми массивами. Библиотека NumPy

Иногда появляется необходимость производить вычисления с огромными наборами данных, представленными в виде массивов или таблиц. В Python для этого принято использовать библиотеку NumPy.

Основное назначение NumPy – то, что она предоставляет объект массива, который более эффективен и лучше подходит для математических вычислений, чем стандартный список Python.

Рассмотрим простой пример, иллюстрирующий важные различия между массивами NumPy и списками:

```
>>> # Списки Python
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # Массивы Numpy
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([ 2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6, 8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

Как видите, основные математические операции с массивами ведут себя иначе. В частности, скалярные операции (например, ax * 2 или ax + 10) применяются к массиву поэлементно (в случае с обычным списком нужно было писать цикл и добавлять в цикле 10 к каждому значению списка). Кроме того, математически операции, когда оба операнда являются массивами, применяются к каждому элементу и в результате создается новый массив. Библиотека NumPy просто огромна и можно ей посвятить отдельную книгу. Посетите сайт *http://www.numpy.org* для дополнительной информации.

5.7. Программа "Угадай число"

5.7.1. ПОСТАНОВКА ЗАДАЧИ

Сейчас мы напишем программу "Угадай число", которая будет демонстрировать следующее:

- Работу с генератором случайных чисел
- Использование цикла **while**
- Прерывание итерации

Работа с циклами была рассмотрена ранее, а сейчас, так сказать, мы теорию закрепим практикой.

Алгоритм работы будет такой:

- В цикле мы "загадываем" случайное число от 1 до 10
- Затем просим пользователя отгадать это число
- Если число правильное, мы выводим соответствующее сообщение и увеличиваем значение переменной **score**

Также будет показано, как исправить логическую ошибку в программе.

5.7.2. РАБОТА С ГЕНЕРАТОРОМ СЛУЧАЙНЫХ ЧИСЕЛ

Для подключения генератора случайных чисел нужно импортировать модуль **random**:

```
import random
```

Далее нужно вызвать функцию *randint()*, передав ей начальное и конечное значение. Возвращенное случайное число будет лежать в диапазоне между ними:

```
random.randint(1, 10)
```

В модуле **random** также есть функция *randrange()*, возвращающая случайное целое число в промежутке от 0 до преданного в качестве параметра зна-

чения (но, не включая само значение), то есть вызов randrange(10) вернет числа от 0 до 9 включительно.

Как по мне, то проще использовать *randint()*, чем *randrange()*. Но это смотря, что вам нужно.

5.7.3. КОД ПРОГРАММЫ

Код программы, действительно, очень прост (лист. 5.1).

Листинг 5.1. Код программы "Угадай число"

```
import random

print("*" * 10, "Угадай число", "*" * 10)

print("Компьютер выберет случайным образом число от 1 до 10.
Попробуй угадать это число. Для выхода введите 0")

answer = 1;
score = 0;
i = 0

while answer:
    i = i + 1
    rand = random.randint(1, 10)
    answer = int(input("Введите число: "))
    if answer == rand:
        score = score +1
        print("Правильно! Ваш счет: ", score, " из ", i)
    else:
        print("Попробуйте еще раз!")

print("До встречи!")
```

Программа ничего сверхъестественного не делает. В цикле **while** она проверяет введенное пользователем значение. Если оно совпадает со сгенерированным в начале итерации случайным значением, значит, выводится соответствующее сообщение и увеличивается значение переменной **score**. Параллельно мы ведем счетчик итераций, чтобы знать, сколько попыток совершил пользователь (переменная **i**).

Посмотрим на вывод программы:

```
********** Угадай число **********
```

```
Компьютер выберет случайным образом число от 1 до 10. Попробуй
угадать это число. Для выхода введите 0
Введите число: 9
Попробуйте еще раз!
Введите число: 8
Правильно! Ваш счет:  1  из  2
Введите число: 5
Правильно! Ваш счет:  2  из  3
Введите число: 2
Попробуйте еще раз!
Введите число: 0
Попробуйте еще раз!
До встречи!
```

5.7.4. ИСПРАВЛЕНИЕ ЛОГИЧЕСКОЙ ОШИБКИ В ПРОГРАММЕ

Все бы хорошо, но в нашей программе есть одна логическая ошибка и как минимум одна недоработка. Для выхода пользователь должен ввести 0. Но посмотрите, что происходит при этом.

Программа считает 0 ... еще одним вариантом, но никак не признаком выхода, поэтому она сообщает, что введенный вариант неправильный. Но он и не может быть правильным, поскольку случайные числа генерируются в диапазоне от 1 до 100.

Исправить эту ошибку можно, если добавим конструкцию:

```
if answer == 0:
  break
```

Данную инструкцию нужно добавить в самое начало тела цикла. Если пользователь введет 0, выполнение будет прервано. Также было бы неплохо, чтобы программа выводила статистику по окончанию игры:

```
print("Общий счет: ", score, " из ", i)
```

Измененный код приведен в листинге 5.2.

Листинг 5.2. Окончательный вариант

```
import random

print("*" * 10, "Угадай число", "*" * 10)
```

```
print("Компьютер выберет случайным образом число от 1 до 10.
Попробуй угадать это число. Для выхода введите 0")

answer = 1;
score = 0;
i = 0

while answer:

    rand = random.randint(1, 10)
    answer = int(input("Введите число: "))

    if answer == 0:
      break
    if answer == rand:
        score = score +1
        print("Правильно!")
    else:
        print("Попробуйте еще раз!")
    i = i + 1

print("Общий счет ", score, " из ", i)
print("До встречи!")
```

Вывод программы будет следующим:

```
********** Угадай число **********
Компьютер выберет случайным образом число от 1 до 10. Попробуй
угадать это число. Для выхода введите 0
Введите число: 7
Попробуйте еще раз!
Введите число: 5
Попробуйте еще раз!
Введите число: 4
Попробуйте еще раз!
Введите число: 0
Общий счет  1  из  3
До встречи!
```

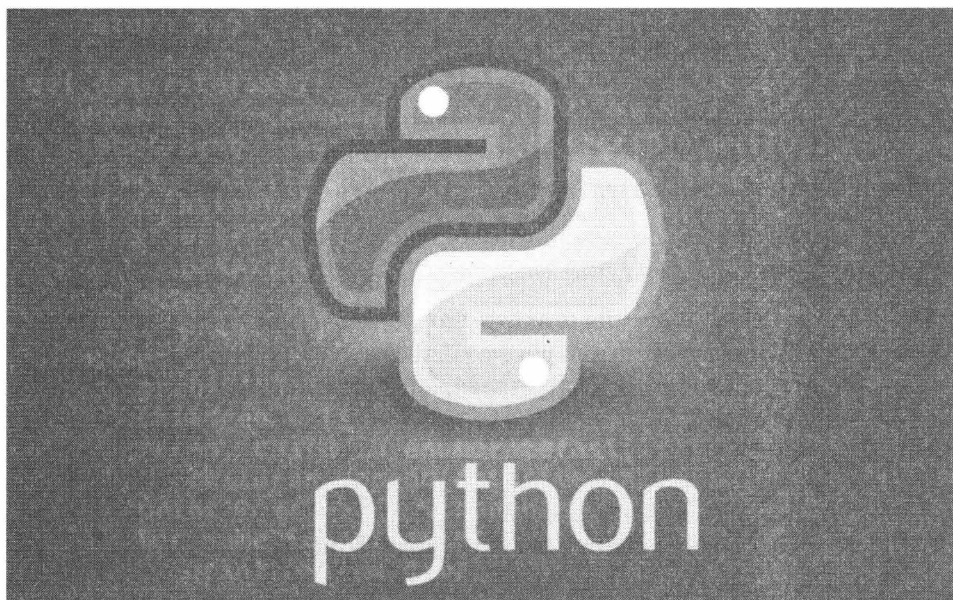
Вот теперь все правильно и работает как нужно!

Примечание. При чтении данных мы не производим проверку их корректности. Если пользователь введет строку вместо числа, то выполнение программы будет установлено, а на консоли

будет изображено сообщение об ошибке. Для обработки таких ситуаций используются блоки try..except, которые мы пока рассматривать не будем.

ГЛАВА 6.

СТРОКИ И СТРОКОВЫЕ ФУНКЦИИ



6.1. Что такое строка? Выбор кавычек

Строка – это *упорядоченная последовательность символов*. Можно даже сказать, что строка – это массив символов, поскольку массив – это и есть упорядоченная последовательность.

Строки поддерживают обращение по индексу, конкатенацию (+), повторение (*), проверку на вхождение (in).

В Python строковые значения принято заключать в кавычки – двойные или одинарные. Компьютеру все равно, главное, чтобы использовался один и тот же тип открывающейся и закрывающейся кавычки, например:

```
print("Привет")
print('Привет')
```

Эти операторы выведут одну и ту же строку. При желании можно, чтобы строка содержала кавычки обоих типов:

```
print("Привет, 'мир'!")
```

Здесь внешние кавычки (двойные) используются для ограничения строкового значения, а внутренние выводятся как обычные символы. Внутри этой строки вы можете использовать сколько угодно одинарных кавычек.

Можно поступить и наоборот – для ограничения использовать одинарные кавычки, тогда внутри можно будет использовать сколько угодно двойных кавычек:

```
print('Привет, "мир"!')
```

Использовав кавычки одного типа в роли ограничителей, вы уже не сможете пользоваться ими внутри строки. Это целесообразно, ведь второе по порядку вхождение открывающей кавычки компьютер считает концом строки.

Функции print() можно передать несколько значений, разделив их запятыми:

```
print("Привет",
      "мир!")
```

Иногда такой прием используют, чтобы сделать код более читабельным.

Если вы внимательно читали предыдущие главы, то знаете, что строки являются неизменяемыми типами данных. Именно поэтому почти все строковые методы в качестве значения возвращают новую строку, а не изменяют существующую. С одной стороны, это хорошо – вам не нужно беспокоиться, что что-то пойдет не так. С другой, при работе с большими объемами данных можно столкнуться с нехваткой памяти.

Помните, что вы можете получить символ строки по индексу, но изменить строку, то есть изменить этот символ, как можно было в других языках программирования, нельзя:

```
>>> str = "Hello"
>>> str[1]
'e'
>>> str[1] = "r"
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    str[1] = "r"
TypeError: 'str' object does not support item assignment
>>>
```

Python поддерживает следующие строковые типы: **str**, **bytes** и **bytearr**. Первый тип – это обычная Unicode-строка. Символы хранятся в некоторой абстрактной кодировке, а при выводе вы можете указать нужную вам кодировку с помощью метода *encode()*:

```
>>> s = "Привет"
>>> s.encode(encoding="utf-8")
b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
>>> s.encode(encoding="cp1251")
b'\xcf\xf0\xe8\xe2\xe5\xf2'
```

Тип **bytes** – это неизменяемая последовательность байтов. Каждый элемент такой последовательности может хранить целое число от 0 до 255, обозначающее код символа. Этот тип поддерживает большинство строковых методов, однако при доступе по индексу возвращается целое число, а не символ:

```
>>> s = bytes("hello", "utf-8")
>>> s[0], s[1], s[2]
(104, 101, 108)
>>> s
b'hello'
```

Некоторые строковые функции некорректно работают с типом **bytes**. Например, функция *len()* возвращает количество байтов, которые занимает строка в памяти, а не количество символов:

```
>>> len("hello")
5
>>> len(bytes("Привет", "utf-8"))
12
```

В кодировке UTF-8 для кодирования одного символа используется два байта, поэтому результат – 12, а не 6.

Тип **bytearray** – это изменяемая последовательность байтов. Данный тип аналогичен типу **bytes**, но вы можете изменять элементы такой строки по индексу. Также этот тип содержит дополнительные методы, которые позволяют добавлять и удалять элементы:

```
>>> s = bytearray("hello", "utf-8")
>>> s[0] = 50; s
bytearray(b'2ello')
```

6.2. Создание строки

Создать строку можно, указав ее между апострофами или двойными кавычками, как уже было показано выше:

```
>>> a = "hello"; a
'hello'
>>> b = "hi!"; b
'hi!'
```

Данные строки ничем не отличаются, и вы можете использовать любой способ, какой вам больше нравится. В PHP есть разница между строками, заключенными в кавычки и в апострофы. В Python разницы никакой нет. Мой совет следующий: если строка содержит апострофы, заключайте ее в кавычки, если же строка содержит кавычки, то заключайте ее в апострофы. Все специальные символы в этих строках (что в кавычках, что в апострофах) интерпретируются, например, \t – это символ табуляции, \n – символ новой строки и т.д. Если нужно вывести символ \ как есть, его нужно экранировать:

```
>>> s = "Hello\\nworld"; s
'Hello\\nworld'
```

При использовании кавычек и апострофов вы не можете разместить объект на нескольких строках. Если нужно присвоить переменной многострочный текст, используйте тройные апострофы:

```
>>> s = '''Hello,
****
world! **** '''
```

Также создать строку можно с помощью функции *str()*:

```
str(<строка>, <кодировка>, <обработка ошибок>)
```

Преимущество этой функции – вы сразу можете указать кодировку, в которой находится текст:

```
>>> s = str(b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\
x82', 'utf-8'); s
'Привет'
```

Обратите внимание: ранее мы использовали пример, в котором мы переменной **str** присваивали значение. Этим мы переопределили идентификатор **str** и при вызове функции **str** вы можете получить сообщение:

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    s = str(b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\
x82', 'utf-8');
TypeError: 'str' object is not callable
```

По привычке многие из нас используют идентификатор **str** для хранения какой-то промежуточной строки. В отличие от PHP, где можно использовать переменную $str и функцию *str()*, в Python этого лучше не делать, иначе вы не сможете использовать функцию *str()*. Да, PHP более гибкий язык и в нем вы легко можете использовать следующий код:

```
<?php
   function str ($str) {
       echo $str;
   }
   $str = "Hello";
   str($str);
?>
```

Одни сплошные идентификаторы **str**, но в Python так делать нельзя. Зато в Python есть строки документирования, которые сохраняются в атрибуте __doc__. Пример:

```
>>> def func1():
   """ Краткое описание """
   pass

>>> print(func1._doc_)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print(func1._doc_)
AttributeError: 'function' object has no attribute '_doc_'
>>> print(func1.__doc__)
 Краткое описание
>>>
```

Обратите внимание, что имя атрибута содержит четыре знака подчеркивания – два до **doc** и два после, иначе вы получите ошибку.

Перед некоторыми строками необходимо разместить модификатор **r**. Специальные символы внутри строки будут выводиться как есть, например, \t не будет преобразован в символ табуляции. Такой модификатор будет полезен, если вы работаете с регулярными выражениями, а также при указании пути к файлу и каталогу:

```
>>> print(r"c:\test\test.py")
c:\test\test.py
```

Если модификатор не указать, то слеши нужно экранировать:

```
>>> print("c:\\test\\test.py")
c:\test\test.py
```

6.3. Тройные кавычки

Иногда есть большой фрагмент текста, который нужно вставить в программу как есть, и вывести в неизменном виде. Конечно, для этого лучше использовать файлы – записать текст в файл, потом в программе прочитать текст из файла и вывести его. Но не все программисты хотят усложнять программу – и если программа несложная, то можно весь код хранить в одном файле, чтобы ничего не потерялось.

Для вывода текста как есть используются тройные кавычки. В листинге 6.1 мы рассмотрим небольшую программу, выводящую инструкцию по использовании абстрактной программы. Когда мы будем изучать функции, данный вывод можно будет оформить в отдельную функцию, а пока разберемся, как работают тройные кавычки.

Листинг 6.1. Вывод многострочного текста прямо из программы

```
print("""
Использование: program -if <input file> [-of <output file>]

-if: входной файл
-of: результирующий файл. Если не указан, будет использован
стандартный вывода
```

```
""")
# Ждем, пока пользователь нажмет Enter
input("\Нажмите Enter для выхода\n") # Это тоже комментарий
```

Текст, заключенный между парой тройных кавычек (""" текст """) выводится, как есть – сохраняется форматирование, переносы строк и т.д. Тройные кавычки существенно облегчают вывод многострочного текста.

6.4. Специальные символы

Внутри строк в Python можно использовать специальные символы, то есть комбинации символов, которые обозначают служебные или непечатаемые символы, которые нельзя вставить обычным способом. Наверняка, вам знакомы эти символы по другим языкам программирования (см. табл. 6.1).

Таблица 6.1. Специальные символы

Специальный символ	Что представляет
\r	Возврат каретки
\n	Перевод строки
\t	Табуляция
\v	Вертикальная табуляция
\b	Забой
\f	Перевод формата
\a	Звонок (BELL)
\0	Нулевой символ
\'	Апостроф
\"	Кавычка
\n	n – восьмеричное значение (код символа), например, \50 (символ 2)

\xn	n – hex-значение символа (код символа), например, \x5b соответствует символу [
\\	Обратный слеш
\uxxxx	16-битный символ Unicode
\Uxxxxxxxx	32-битный символ Unicode

6.5. Действия над строками

Строки поддерживают следующие операции:

- Обращение к элементу по индексу;
- Срез;
- Конкатенация;
- Проверку на вхождение;
- Повтор.

6.5.1. ОБРАЩЕНИЕ К ЭЛЕМЕНТУ ПО ИНДЕКСУ

Ранее было показано, как обратиться к отдельному символу строки. Нумерация символов начинается с 0:

```
>>> s = "123"
>>> s[0], s[1], s[2]
('1', '2', '3')
```

Если обратиться к несуществующему символу строки, получите следующую ошибку:

```
>>> s[3]
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    s[3]
IndexError: string index out of range
```

Вы можете указать отрицательное значение индекса. В этом случае отсчет будет с конца строки:

```
>>> s[-1], s[-2], s[-3]
('3', '2', '1')
```

6.5.2. СРЕЗ СТРОКИ

Очень интересной является операция среза строки. Ее формат следующий:

```
[<start>:<end>:<step>]
```

Интересна она хотя бы даже тем, что все три параметра являются необязательными. Например, если не указан параметр <start>, то по умолчанию будет использовано значение 0. Если не указан параметр <end>, то будет возвращен фрагмент до конца строки. И, если не указан <step>, будет использоваться шаг 1. В качестве всех трех параметров можно указать отрицательные значения.

```
>>> s = "Hello"
>>> s[:]        # Фрагмент от позиции 0 до конца строки
'Hello'
>>> s[:-1:]       # Отрезаем последний символ строки
'Hell'
>>> s[1::2]       # Начиная с позиции 1 до конца строки, шаг 2
'el'
>>> s[1::]        # Отрезаем первый символ
'ello'
```

Поэкспериментируйте с операцией среза – я уверен, что вам она понравится.

6.5.3. КОНКАТЕНАЦИЯ СТРОК

Конкатенация строк бывает явной и неявной. *Явная* – это использование оператора +, а *неявная* – это указание двух или более строк рядом – через пробел:

```
>>> print("1" + "2")
12
>>> print("1" "2")
12
```

Преобразовать несколько строк в кортеж можно с помощью запятой, например:

```
>>> s = "1", "2"
>>> type(s)
<class 'tuple'>
```

Как видите, мы получили тип **tuple** – кортеж, а не строку. Вот только помните, что вы не можете выполнить неявную конкатенацию строки и переменной, например:

```
>>> print("3" s)
SyntaxError: invalid syntax
```

6.5.4. ПРОВЕРКА НА ВХОЖДЕНИЕ

Проверить, входит ли подстрока в строку, можно с помощью оператора **in**:

```
>>> "hell" in "Hello"
False
>>> "hell" in "hello"
True
```

Оператор **in**, как вы уже успели заметить, чувствителен к регистру символов.

6.5.5. ПОВТОР

Оператор * позволяет повторить строку определенное число раз, например:

```
>>> print("*" * 20)
********************
```

6.5.6. ФУНКЦИЯ LEN()

Функция *len()* возвращает количество символов в строке. Напомню, что с байтовыми строками эта функция работает некорректно и возвращает количество байтов, которые занимает строка:

```
>>> len("123456")
6
```

Функцию *len()* можно использовать для перебора всех символов строки:

```
>>> s = "123456"
>>> for i in range(len(s)): print(s[i], end=" ")

1 2 3 4 5 6
```

Помните, что в случае с Unicode-строками количество байтов, необходимых для хранения символов строки, превышает само число символа, и вы можете получить ошибку выхода за пределы диапазона.

6.6. Форматирование строки и метод format()

6.6.1. ОПЕРАТОР ФОРМАТИРОВАНИЯ %

Программистам, знающим язык С, знакомая функция *printf()*, выводящая строку в определенном формате. Язык Python также поддерживает форматирование строки. На данный момент в Python поддерживается два способа форматирования текста:

- Оператор %
- Метод format()

В следующей версии Python оператор % могут удалить, поэтому настоятельно рекомендуется использовать метод *format()*. Но не рассмотреть, хотя бы вкратце, оператор % мы не можем, поскольку все еще есть множество кода, написанного с использованием этого оператора. Формат оператора % следующий:

```
<Формат> % <Значения>
```

Синтаксис описания формата такой:

```
%[(<Ключ>)][<Флаг>][<Ширина>][.<Точность>]<Преобразование>
```

Пример использования оператора форматирования:

```
>>> "%s/%s/%s" % (30, 10, 2020)
'30/10/2020'
```

Рассмотрим параметры формата. Первый параметр – <Ключ>. Он задает ключ словаря, если он задан, то в параметре <Значения> нужно указать словарь, а не кортеж. Вот пример:

```
>>> "%(car)s - %(year)s" % {"car" : "nissan", "year" : 2021}
'nissan - 2021'
```

Параметр <Флаг> – это флаг преобразования, который может содержать

следующие значения:

- **#** — для восьмеричных значений добавляет в начало символы 0o, для шестнадцатеричных – 0x, для вещественных чисел – будет выводиться точка, даже если дробная часть равна 0
- **0** — если указан, будут выводиться ведущие нули для числового заполнения
- **-** — задает выравнивание по левой границе области
- **пробел** — добавляет пробел перед положительным числом, перед отрицательным будет выводиться -
- **+** — обязательный вывод знака, как для отрицательных, так и для положительных чисел

Примеры:

```
>>> "%#x %#x" % (0xfff, 100)
'0xfff 0x64'
>>> "%+d %+d" % (-3, 3)
'-3 +3'
```

Параметр <Ширина> определяет минимальную ширину поля, но если

строка не помещается в указанную ширину, то значение будет проигнорировано и строка будет выведена полностью. Пример:

```
>>> "'%10d' - '%-10d'" % (5, 5)
"'         5' - '5         '"
```

Параметр <Точность> задает количество знаков после точки для вещественных чисел:

```
>>> from math import *
>>> "%s %f %.2f" % (e, e, e)
'2.718281828459045 2.718282 2.72'
```

Последний параметр <Преобразование> является обязательным и может содержать следующие значения:

- **a** — пытается преобразовать любой объект в строку, используя функцию ascii();
- **c** — выводит одиночный символ или преобразует число в символ;
- **d (i)** — возвращает целую часть числа;
- **e** — вещественное число в экспоненциальной форме (буква "e" в нижнем регистре);
- **E** — вещественное число в экспоненциальной форме (буква "e" в верхнем регистре);
- **f (F)** — используется для вывода вещественного числа;
- **g** — то же самое, что f или E (используется более короткая запись числа);
- **G** — то же самое, что F или E (используется более короткая запись числа);
- **s** – пытается преобразовать любой объект в строку (с помощью функции str());
- **r** — то же, что и **s**, но для преобразования в строку вместо функции str() будет использоваться функция repr();
- **o** — выводит восьмеричное значение;
- **x** — шестнадцатеричное значение в нижнем регистре;
- **X** — шестнадцатеричное значение в верхнем регистре.

Ранее было продемонстрировано использование модификаторов d, f, s, x. Остальные модификаторы используются аналогично. Вместо множества примеров, которые вы и сами можете провести, я подскажу, как правильно нужно использовать модификаторы формата и оператор %.

Представим, что у нас есть HTML-шаблон, который нужно заполнить данными. В этом случае идеально подходит оператор % (лист. 6.2).

Листинг 6.2. Правильное использование оператора %

```
template = """<html>
<head>
  <title>%(title)s</title>
</head>
<body>
%(text)s
</body>
</html>"""

data = { "title": "Мой сайт",
         "text": "Контент"}

print(template % data)
```

Переменная **template** содержит код шаблона, а переменная **data** – данные шаблона. Затем последним оператором мы заполняем наш шаблон данными. Результат изображен на рис. 6.1.

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========================= RESTART: C:\temp\py\6-2.py =========================
<html>
<head>
  <title>Мой сайт</title>
</head>
<body>
Контент
</body>
</html>
>>> |
Ln: 13 Col: 4
```

Рис. 6.1. Результат работы программы из листинга 6.2.

6.6.2. МЕТОДЫ ВЫРАВНИВАНИЯ СТРОКИ

Прежде, чем перейти к рассмотрению метода *format()*, нужно рассмотреть дополнительные методы, которые вы можете использовать для выравнивания строки:

- *center(<Ширина>[, <Символ>])* – выравнивание строки по центру поля указанной ширины. Второй параметр задает символ, который будет добавлен слева и справа от указанной строки. По умолчанию второй параметр равен пробелу.
- *ljust(<Ширина>[, <Символ>])* – выравнивание по левому краю.
- *rjust(<Ширина>[, <Символ>])* – по правому краю.

Примеры:

```
>>> s = "Hello"
>>> s.center(20)
'       Hello        '
>>> s.center(20, '*')
'*******Hello********'
>>> s.ljust(20)
'Hello               '
>>> s.rjust(20)
'               Hello'
```

6.6.3. МЕТОД FORMAT()

Метод *format()* используется для выравнивания строк, начиная с версии Python 2.6. Пока еще оператор % оставлен из соображения обратной совместимости с тоннами уже написанного кода, но в скором времени останется лишь метод format().

Использовать этот метод нужно так:

```
<Строка> = <Формат>.format(*args, **kwargs)
```

Синтаксис строки формат следующий:

```
{[<Поле>][!<Функция>][:<Формат>]}
```

Начнем с параметра <Поле>. В нем можно указать индекс позиции или ключ. Помните, что нумерация начинается с 0. Также можно комбинировать

именованные и позиционные параметры. В этом случае в методе format() именованные параметры используются в конце:

```
>>> "{0}/{1}/{2}".format(30, 10, 2020)
'30/10/2020'
>>> "{0}/{1}/{2}".format(*arr)
'30/10/2020'
>>> "{firstname} {lastname}".format(firstname="Иван",
lastname="Петров")
'Иван Петров'
>>> "{lastname}, {0}".format("Иван", lastname="Петров")
'Петров, Иван'
```

Параметр <Функция> позволяет задать функцию, с помощью которой обрабатываются данные перед их вставкой в строку. Если указано значение "s", то данные обрабатываются функцией *str()*, если указано значение "r" – функцией *repr()*, если "a", то – *ascii()*. По умолчанию используется функция *str()*.

В параметре <Формат> указывается значение, имеющее формат:

```
[[<Заполнитель>]<Выравнивание>][<Знак>][#][0][<Ширина>][,]
[.<Точность>][<Преобразование>]
```

По умолчанию выравнивание выполняется по правому краю, но вы можете изменить это поведение с помощью параметра <Выравнивание>. Вы можете указать следующие значения:

- < — по левому краю;
- > — по правому краю;
- ^ — по центру;
- = — знак числа выравнивается по левому краю, а само число – по правому.

Пример:

```
>>> "'{0:<15}' '{1:>15}'".format('Hello', 'Hello')
"'Hello          ' '          Hello'"
```

По умолчанию <Заполнитель> равен пробелу, но вы можете указать любой другой символ, например:

```
>>> "'{0:*<15}' '{1:&>15}'".format('Hello', 'Hello')
"'Hello**********' '&&&&&&&&&&Hello'"
```

Параметр <Знак> управляет отображением знака числа:

- **+** — знак будет отображаться, как для положительных, так и для отрицательных чисел;
- **-** — знак будет выводиться только для отрицательных чисел (по умолчанию);
- **пробел** — вместо **+** для положительных чисел будет выводиться **пробел**, для отрицательных – **минус**.

Как и в случае с оператором %, параметр <Ширина> задает минимальную ширину поля. Если строка не помещается в указанную ширину, то значение будет проигнорирована и строка будет выведена полностью.

Переходим к самому важному параметру – <Преобразование>. Для целых чисел можно использовать следующие модификаторы формата:

- **b** — двоичное значение;
- **c** — преобразует целое число в соответствующий ему символ;
- **d** — десятичное значение;
- **n** — выводит десятичное значение с учетом настроек локали;
- **o** — восьмеричное значение;
- **x** — шестнадцатеричное значение в нижнем регистре;
- **X** — шестнадцатеричное значение в верхнем регистре.

Для вещественных чисел можно использовать эти модификаторы:

- **f** и **F** — вещественное число;
- **e** — вещественное число в экспоненциальной форме (буква "e" в нижнем регистре);
- **E** — вещественное число в экспоненциальной форме (буква "e" в верхнем регистре);
- **g** — то же самое, что f или E (используется более короткая запись числа);
- **G** — то же самое, что F или E (используется более короткая запись числа);

- **n** — аналогично g, но с учетом настройки локали;
- **%** — умножает число на 100 и добавляет символ процента в конце.

Пример:

```
>>> "'{0:G}' '{1:e}'".format(pi, pi)
"'3.14159' '3.141593e+00'"
```

6.7. Функции и методы для работы со строками

В Python очень много функций и методов для работы со строками. Данная книга не является справочным руководством по Python (хорошо, что такое руководство легко найти в Интернете), поэтому мы рассмотрим только основные функции и методы (табл. 6.1 и табл. 6.2). Некоторые из этих методов мы рассмотрим в этом разделе, некоторые – далее в этой главе.

Таблица 6.1. Функции для работы со строками

Синтаксис	Описание
str([<объект>])	Преобразует любой объект в строку. Если параметр не указан, то возвращается пустая строка. Данная функция используется функцией print() и некоторыми другими функциями для вывода объектов
repr(<объект>)	Возвращает строковое представление объекта. Используется в IDLE для вывода объектов
ascii(<объект>)	Возвращает строковое представление объекта, при выводе объекта используются только ASCII-символы
len(<Строка>)	Возвращает количество символов в строке

Примеры:

```
>>> str("Hello")
```

```
'Hello'
>>> repr("Hello")
"'Hello'"
>>> ascii("Hello")
"'Hello'"
>>> ascii("Привет")
"'\\u041f\\u0440\\u0438\\u0432\\u0435\\u0442'"
>>> len("Привет")
6
```

Таблица 6.2. Строковые методы (основные)

Синтаксис	Описание
capitalize()	Делает прописной первую букву строки
center(width, [fill])	Возвращает отцентрованную строку, по краям которой стоит символ fill (пробел по умолчанию)
count(str, [start],[end])	Возвращает количество непересекающихся вхождений подстроки в диапазоне [начало, конец] (0 и длина строки по умолчанию)
endswith(str)	Заканчивается ли строка S шаблоном str
expandtabs([tabsize])	Возвращает копию строки, в которой все символы табуляции заменяются одним или несколькими пробелами, в зависимости от текущего столбца. Если TabSize не указан, размер табуляции полагается равным 8 пробелам
find(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или -1
format(*args, **kwargs)	Форматирование строки

index(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или вызывает ValueError
isalnum()	Состоит ли строка из цифр или букв. Возвращает true, если это так
isalpha()	Состоит ли строка из букв
isdigit()	Состоит ли строка из цифр
islower()	Состоит ли строка из символов в нижнем регистре
isspace()	Состоит ли строка из неотображаемых символов (пробел, символ перевода страницы ('\f'), "новая строка" ('\n'), "перевод каретки" ('\r'), "горизонтальная табуляция" ('\t') и "вертикальная табуляция" ('\v'))
istitle()	Начинаются ли слова в строке с заглавной буквы
isupper()	Состоит ли строка из символов в верхнем регистре
join(<Последовательность>)	Преобразует последовательность в строку. Элементы разделяются через указанный разделитель. Метод используется так: `<Строка> = <Разделитель>.join(<Послед-ть>)`
ljust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя последние символы символом *fillchar*
lower()	Переводит все символы строки в нижний регистр

lstrip([<Символы>])	Удаляет указанные или пробельные символы в начале строки
partition(<Разделитель>)	Находит первое вхождение разделителя в строку и возвращает кортеж из трех элементов: `Фрагмент до разделителя` `Символ-разделитель` `Фрагмент после разделителя`
replace(old, new [,max])	Возвращает строку, в которой вхождения строки *old* заменены строкой *new*. Необязательный параметр max устанавливает наиболее возможное количество замен
rfind(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
rindex(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или вызывает *ValueError*
rjust(width, fillchar=" ")	Делает длину строки не меньшей *width*, по необходимости заполняя первые символы символом *fillchar*
rpartition(<Разделитель>)	То же, что и partition, но поиск разделителя осуществляется справа налево
rsplit([<Разделитель>[, <Лимит>]])	То же, что и *split*, но поиск разделителя осуществляется справа налево
rstrip([<Символы>])	Удаляет указанные или пробельные символы в конце строки

split([<Разделитель>[, <Лимит>]])	Разделяет строку на подстроки по указанному разделителю. Если не указан первый параметр (или передано значение *None*), то вместо разделителя используется пробел. Второй параметр <Лимит> ограничивает количество подстрок
startswith(str)	Начинается ли строка S с шаблона str
strip([<Символы>])	Удаляет указанные символы в начале и конце строки. Если символы не указаны, удаляются пробельные символы: пробел, табуляция (\t, \v), возврат каретки (\r), перенос строки (\n)
swapcase()	Своп-регистра: верхний регистр заменяется на нижний и наоборот
title()	Делает прописной первую букву каждого слова
upper()	Переводит все символы в верхний регистр
zfill(width)	Делает длину строки не меньшей width, по необходимости заполняя первые символы нулями

Примеры:

```
# Обрезаем пробельные символы
>>> s = "  Hello  \n"
>>> s.strip()
'Hello'
# Методы работают с копией строки, поэтому исходная строка
# не изменяется
>>> s
'  Hello  \n'
>>> s.lstrip()
'Hello  \n'
>>> s.rstrip()
'  Hello'
```

```
# Метод split() без параметров
>>> s = "поле1 поле2 поле3"
>>> s.split()
['поле1', 'поле2', 'поле3']
# Параметры None, 1
>>> s.split(None, 1)
['поле1', 'поле2 поле3']

# Метод partition()
>>> s.partition(" ")
('поле1', ' ', 'поле2 поле3')
>>> s.rpartition(" ")
('поле1 поле2', ' ', 'поле3')

# Пример использования join()
>>> sep = " "
>>> s1 = sep.join(["поле1", "поле2"])
>>> s1
'поле1 поле2'

# Функции изменения регистра
>>> s = "hello, world"
>>> s.capitalize()
'Hello, world'
>>> s.title()
'Hello, World'
>>> s.upper()
'HELLO, WORLD'
>>> s.lower()
'hello, world'
>>> s.swapcase()
'HELLO, WORLD'
```

Также вы можете использовать функции *chr()* и *ord()*. Наверняка вы знакомы с ними, если знаете другие языки программирования. Первая возвращает символ, соответствующий заданному коду, а вторая – возвращает код символа:

```
>>> chr(55)
'7'
>>> ord('7')
55
>>> chr(5055)
'Ꮏ'
```

6.8. Настройка локали

Локаль – это *совокупность локальных настроек системы* (формат даты, формат времени, кодировка, форматирование денежных единиц и чисел и т.д.). Для установки локали используется функция *setlocale()* из модуля **locale**.

Синтаксис функции следующий:

```
setlocale(<категория>, <локаль>)
```

Категория задает категорию настроек, которые будут изменены после вызова *setlocale()*:

- LC_ALL с все настройки;
- LC_MONETARY – для денежных единиц;
- LC_TIME — настройки, влияющие на форматирование даты и времени;
- LC_NUMERIC — настройки, влияющие на форматирование чисел.

Пример:

```
import locale
locale.setlocale(locale.LC_ALL, ('russian'))
```

6.9. Поиск и замена в строке

Для поиска и замены в строке в Python используется множество методов. Начнем с метода *find()*, формат которого выглядит так:

```
<строка>.find(<подстрока>[, <начало>[, <конец>]])
```

Метод осуществляет поиск подстроки в строке. Последние два параметры необязательны. Если они указаны, то производится операция среза строки:

```
<Строка>[<Начало>:<Конец>]
```

Метод возвращает номер позиции, с которой начинается вхождение подстроки в строку:

```
>>> s = "hello, world"
>>> num = s.find("world")
>>> num
7
```

Если подстрока не входит в строку, возвращает -1.

Метод *index()* похож на метод *find()*, но возвращает исключение *ValueError*, если подстрока в строку не входит.

Следующий метод – *rfind()*:

```
<строка>.rfind(<подстрока>[, <начало>[, <конец>]])
```

Похож на метод *find()*, но возвращает позицию последнего вхождения подстроки в строку или -1, если подстрока не входит в строку.

Метод *count()* возвращает число вхождения подстроки в строку. Если подстроки нет в строке, метод возвращает 0. Метод зависит от регистра символов. Синтаксис метода такой:

```
<строка>.count(<подстрока>[, <начало>[, <конец>]])
```

Метод *startswith()* возвращает *True*, если подстрока есть в строке, или же *False*, если подстрока не входит в строку. Синтаксис:

```
<строка>.startswith(<подстрока>[, <начало>[, <конец>]])
```

Метод *endswith()* возвращает *True*, если строка заканчивается указанной подстрокой. Если подстроки нет в строке, возвращается *False*:

```
>>> s.startswith('world')
False
>>> s.endswith('world')
True
```

Метод *replace()* реализует замену всех вхождений подстроки в строку на другую подстроку. Замена:

```
<Строка>.replace(<Заменяемая подстрока>, <Новая подстрока>[,
<Лимит>])
```

Параметр <Лимит> необязательный и ограничивает максимальное количество замен.

```
>>> s.replace('world', 'reader!')
'hello, reader!'
```

6.10. Что в строке?

В Python есть ряд методов, позволяющих проверить тип содержимого строки. Вы можете использовать их для проверки ввода пользователя (табл. 6.3). Аналогичные методы есть в других языках программирования, например, в PHP.

Таблица 6.3. Методы проверки типа содержимого строки

Синтаксис	Описание
isdigit()	Возвращает *True*, если строка состоит только из цифр
isdecimal()	Возвращает *True*, если строка содержит только десятичные символы
isnumeric()	Возвращает *True*, если строка содержит только числовые символы. К числовым символам относятся не только десятичные цифры, но и дробные числа, римские числа и т.д.
isalpha()	Метод возвращает *True*, если строка содержит только буквы
isspace()	*True*, если строка состоит только из пробельных символов
isalnum()	*True*, если строка содержит только буквы и/или цифры. В противном случае и для пустой строки возвращается False
islower()	Возвращает *True*, если строка содержит все символы в нижнем регистре
isupper()	Возвращает *True*, если строка содержит все символы в верхнем регистре

Все эти методы возвращают *False*, если строка не прошла соответствующую проверку. Например:

```
>>> age = input('Ваш возраст: ')
Ваш возраст: 37
>>> if age.isdigit() == True:
  print('Ok')
else:
  print('Ошибка')

Ok
```

6.11. Шифрование строк

Модуль **hashlib** содержит функции, позволяющие зашифровать строки.

Этот модуль очень и очень полезный. Пароли пользователей принято хранить в базе данных в зашифрованном виде. Чтобы не изобретать колесо заново, вы можете использовать модуль hashlib, содержащий функции md5(), sha1(), sha256(), sha384, sha512(). Названия функций соответствуют алгоритмам шифрования.

Пример шифрования пароля с использованием алгоритма MD5:

```
>>> import hashlib
>>> hash = hashlib.md5(b"secret")
>>> hash.digest()
b'^\xbe"\x94\xec\xd0\xe0\xf0\x8e\xabv\x90\xd2\xa6\xeei'
>>> hash.hexdigest()
'5ebe2294ecd0e0f08eab7690d2a6ee69'
```

В базе данных сохраняется, как правило, значение, возвращаемое методом *hexdigest()*.

6.12. Переформатирование текста. Фиксированное число колонок

Для переформатирования текста для вывода используется модуль **textwrap**. Например, представим, что у нас есть следующие строки:

```
s = " Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Quisque sit amet diam quis risus interdum sollicitudin. In
vestibulum, libero id maximus lacinia, leo massa pharetra mi,
```

```
vitae posuere libero dui nec lectus. Ut ac nunc sagittis,
consequat eros eget, dignissim ex. Vivamus ac nisl felis. In
accumsan tristique aliquet. Morbi eu varius urna. Lorem ipsum
dolor sit amet, consectetur adipiscing elit. Curabitur nec
urna sit amet libero tempor rhoncus. Nulla fringilla erat sit
amet augue laoreet volutpat. Etiam quis molestie risus. Morbi
sapien nisi, scelerisque sed finibus non, placerat sed elit.
Sed placerat turpis ac varius tincidunt. Morbi turpis metus,
molestie eu erat eu, ultricies sollicitudin lorem."
```

Далее вы можете использовать модуль **textwrap** для переформатирования текста различными способами:

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Quisque sit
amet diam quis risus interdum sollicitudin. In vestibulum,
libero id
maximus lacinia, leo massa pharetra mi, vitae posuere libero
dui nec
lectus. Ut ac nunc sagittis, consequat eros eget, dignissim
ex.
Vivamus ac nisl felis. In accumsan tristique aliquet. Morbi eu
varius
urna. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Curabitur nec urna sit amet libero tempor rhoncus. Nulla
fringilla
erat sit amet augue laoreet volutpat. Etiam quis molestie
risus. Morbi
sapien nisi, scelerisque sed finibus non, placerat sed elit.
Sed
placerat turpis ac varius tincidunt. Morbi turpis metus,
molestie eu
erat eu, ultricies sollicitudin lorem.
>>>

>>> print(textwrap.fill(s, 40))
 Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Quisque sit amet diam
quis risus interdum sollicitudin. In
vestibulum, libero id maximus lacinia,
leo massa pharetra mi, vitae posuere
libero dui nec lectus. Ut ac nunc
sagittis, consequat eros eget, dignissim
ex. Vivamus ac nisl felis. In accumsan
```

```
tristique aliquet. Morbi eu varius urna.
Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Curabitur nec urna sit
amet libero tempor rhoncus. Nulla
fringilla erat sit amet augue laoreet
volutpat. Etiam quis molestie risus.
Morbi sapien nisi, scelerisque sed
finibus non, placerat sed elit. Sed
placerat turpis ac varius tincidunt.
Morbi turpis metus, molestie eu erat eu,
ultricies sollicitudin lorem.

>>> print(textwrap.fill(s, 40, initial_indent='  '))
  Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Quisque sit
amet diam quis risus interdum
sollicitudin. In vestibulum, libero id
maximus lacinia, leo massa pharetra mi,
vitae posuere libero dui nec lectus. Ut
ac nunc sagittis, consequat eros eget,
dignissim ex. Vivamus ac nisl felis. In
accumsan tristique aliquet. Morbi eu
varius urna. Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Curabitur
nec urna sit amet libero tempor rhoncus.
Nulla fringilla erat sit amet augue
laoreet volutpat. Etiam quis molestie
risus. Morbi sapien nisi, scelerisque
sed finibus non, placerat sed elit. Sed
placerat turpis ac varius tincidunt.
Morbi turpis metus, molestie eu erat eu,
ultricies sollicitudin lorem.
```

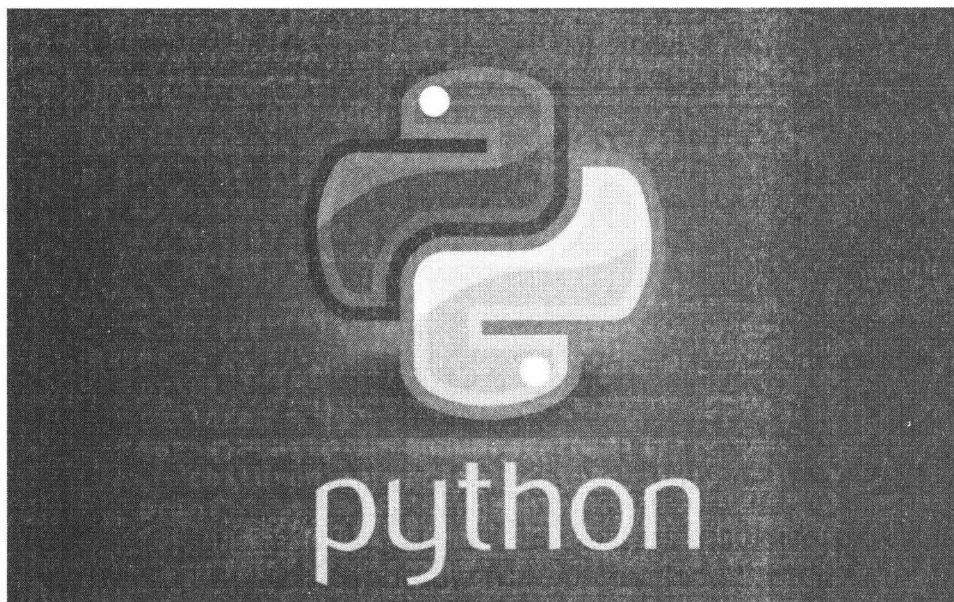
Модуль **textwrap** – простой способ подготовить текст для вывода, особенно, если вы хотите аккуратно вывести его на терминал. Получить количество колонок терминала можно, используя метод *os.get_terminal_size()*. Например:

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

У метода *fill()* есть несколько дополнительных параметров, управляющих табуляцией, окончанием предложений и т.д.

ГЛАВА 7.

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ



7.1. Введение в регулярные выражения

В этой книге мы уже рассмотрели функцию поиска подстроки в строке

– метод *find()*. Реализовать поиск нужной нам строки в файле с помощью этой функции не составит проблем. Все, что нам нужно – это прочитать файл в одну большую строку (благо, размер строки в Python не ограничен, в отличие от некоторых других языков программирования, где размер строки ограничен 255 символами) и вызвать метод *find()*, передав ей искомую строку.

Но наша задача существенно усложняется, если мы знаем, только, как должна выглядеть искомая строка, но мы не знаем точно, какую строку мы ищем. Сейчас поясню, что имеется в виду. Представим, что у нас есть текстовый файл, в котором нужно найти все адреса e-mail. Мы не знаем, какие именно адреса будут в этом файле, поэтому мы не можем указать точную строку для ее передачи методу *find()*. Мы только лишь знаем, как будут выглядеть искомые строки: строка_без_пробелов@имя_домена1.имя_домена2...имя_

доменаN.TLD (TLD – это Top Level Domain – домен верхнего уровня, например, .com, .net и т.д.). Представьте, какой нужно написать алгоритм поиска всех e-mail, реализованный с использованием метода *find()*. Сначала нам нужно найти символ @ – это единственное, что нам известно, этот символ будет в любом e-mail. Затем нам нужно найти позиции пробелов до и после символа @. Затем нужно выделить строки от первого пробела (который находится перед предполагаемым e-mail) и от символа @ до первого пробела, стоящего после него. Затем проанализировать, чем являются эти строки. Например, если длина доменного имени слишком короткая или не содержит точек, то наша строка явно не является e-mail. Одним словом, алгоритм поиска будет слишком сложным.

Намного проще использовать регулярные выражения. Регулярное выражение – это шаблон искомого текста. Если найденный текст соответствует шаблону, говорят, что он соответствует регулярному выражению. При этом ваша программа не будет громоздкой – вам нужно лишь вызвать функцию поиска по регулярному выражению.

Поддержка регулярных выражений содержится в модуле **re**:

```
import re
```

> ***Примечание.*** Ранее модуль, обеспечивающий поддержку регулярных выражений, назывался **regex**, но, начиная с версии 2.5, его поддержка была удалена.

7.2. Функция compile() и основы регулярных выражений

Функция *compile()* позволяет создать откомпилированный шаблон регулярного выражения. Вызывать ее нужно так:

```
<Шаблон> = re.compile(<Регулярное выражение>[, <Модификатор>])
```

Параметр модификатор может содержать следующие флаги:

- A или ASCII – классы \w, \W, \b, \B, \d, \D, \s и \S будут соответствовать обычным символам

- L или LOCALE – учитывает настройки текущей локали
- I или IGNORECASE – поиск без учета регистра
- M или MULTILINE – поиск в строке, которая состоит из нескольких подстрок, разделенных символом новой строки ("\n"). Символ ^ соответствует привязке к началу каждой подстроки, а символ $ – позиции перед символом перевода строки
- S или DOTALL – символ "точка" соответствует любому символу, включая символ перевода строки (\n). По умолчанию "точка" не соответствует символу перевода строки. Символ ^ будет соответствовать привязке к началу строки, а $ – привязке к концу всей строки
- X или VERBOSE – пробелы и символы перевода строки будут игнорированы. Внутри регулярного выражения можно использовать комментарии
- U или UNICODE – классы \w, \W, \b, \B, \d, \D, \s и \S будут соответствовать Unicode-символам. В Python 3 флаг установлен по умолчанию

Рассмотрим несколько примеров:

```
>>> import re
# Игнорируем регистр
>>> p = re.compile(r"^[a-z]+$", re.I)
>>> print("Найдено" if p.search("ABC") else "Not found")
# Найдено, хотя в строке символы в верхнем регистре, а в
шаблоне - в нижнем
Найдено

# Регистр символов не игнорируется
>>> p = re.compile(r"^[a-z]+$")
>>> print("Найдено" if p.search("ABC") else "Не найдено")
# Не найдено
Не найдено

>>> p = re.compile(r"^.$")
>>> print("Найдено" if p.search("\n") else "Не найдено")
Не найдено
```

Посмотрите на примеры – до строки, содержащей регулярное выражение, указывается модификатор **r**. Мы используем неформатированные строки – если модификатор не указывать, то все слэши нужно будет экранировать. Например, строку:

```
p = re.compile(r"^\a+$")
```

нужно будет записать так:

```
p = re.compile("^\\a+$")
```

Внутри регулярного выражения символы ., ^, $, *, +, ?, {, [,), \, 1, (и) имеют специальное значение. Если эти символы должны трактоваться как есть, то их следует экранировать с помощью слэша. Некоторые специальные символы теряют свое особое значение, если их разместить внутри квадратных скобок.

Например, символ "точка" по умолчанию соответствует любому символу, кроме символа перевода строки. Если необходимо найти точку, то перед ней нужно указать \ или разместить точку внутри скобок – [.].

Рассмотрим небольшой пример:

```
>>> dt = "01.01.2021"
>>> p = re.compile(r"^[0-3][0-9].[01][0-9].[12][09][0-9][0-
9]$")
>>> if p.search(dt): print("+")
+
```

В этом примере мы использовали методы ^ и $. Кроме этих символов есть метасимволы \A и \Z. Назначение этих метасимволов следующее:

- ^ — привязка к началу строки или подстроки (зависит от флагов M или S)
- $ — привязка к концу строки или подстроки (зависит от флагов M или S)
- \A — привязка к началу строки (без зависимости от модификатора)
- \Z — привязка к концу строки (без зависимости от модификатора)

Примеры:

```
# Точка не соответствует \n
>>> p = re.compile(r"^.+$")
>>> p.findall("s1\ns2\ns3")
[]
# Точка соответствует \n
>>> p = re.compile(r"^.+$", re.S)
>>> p.findall("s1\ns2\n\s3")
['s1\ns2\n\\s3']
```

Привязка к началу или концу строки используется, только если строка должна полностью соответствовать регулярному выражению. Давайте напишем программу, которая проверяет, является ли строка числом:

```
import re

p = re.compile(r"^[0-9]+$", re.S)

num = "123"
snum = "s123"

if p.search(num):
  print("Число")
else:
  print("Строка")

if p.search(snum):
  print("Число")
else:
  print("Строка")
```

Результат выполнения этого модуля будет следующим:

```
Число
Строка
```

Сначала мы передаем строку, которая содержит только число, поэтому будет выведено *Number*. Далее мы передаем строку, которая не полностью состоит из числа, поэтому будет выведено *String*.

В квадратных скобках указываются символы, которые могут встречаться на этом месте в строке. Диапазон символов перечисляется через тире. Примеры:

- [13] – соответствует числу 1 или 3
- [0-9] – соответствует числу от 0 до 9
- [abc] – соответствует буквам "a", "b" или "c"
- [a-z] – соответствует буквам от "a" до "z"
- [a-zA-Z] – соответствует любой букве английского алфавита
- [0-9a-zA-Z] – любая буква или любая цифра

С помощью символа ^ можно инвертировать значение, указанное в скобках. Вы можете указать символы, которых не должно быть на этом месте в строке, например, [^13] – символов 1 и 3 не должно быть в строке.

Если не хочется указывать символы, то можно указать сразу классы символов (табл. 7.1).

Таблица 7.1. Классы символов

Класс символов	Что означает
\d	Соответствует любой цифре
\w	Соответствует любой букве, цифре и знаку подчеркивания. Эквивалентно [a-zA-Z0-9_] при указании флага A
\s	Любой пробельный символ. При указании флага A эквивалентно [\t\n\r\f\v]
\D	Не цифра. При указании флага A эквивалентно [^0-9]
\W	Не буква, не цифра и не символ подчеркивания. Эквивалентно [^a-zA-Z0-9_]
\S	Не пробельный символ, эквивалентно [^\t\n\r\f\v]

Количество вхождения символа в строку задается с помощью квантификаторов:

- **{n}** — **n** вхождений символа в строку. Например, r"^[0-9]{3}$" соответствует трем вхождениям любой цифры
- **{n,}** — минимум n (n или больше) вхождений символа
- **{n,m}** — от **n** до **m** вхождений
- ***** — ноль или больше вхождений символа в строку. Эквивалентно {0,}
- **+** — одно или больше число вхождений символа в строку. Эквивалентно {1,}
- **?** — ни одного или одно вхождение в строку. Эквивалентно комбинации {0,1}

Регулярные выражения по умолчанию "жадные". То есть ищут самую длинную последовательность, которая соответствует шаблону, и не учитывают более короткие соответствия. Рассмотрим следующий пример:

```
>>> s = "<i>Один</i><i>Два</i><i>Три</i>"

>>> p = re.compile(r"<i>.*</i>", re.S)
>>> p.findall(s)
['<i>Один</i><i>Два</i><i>Три</i>']
```

Как видите, была найдена вся строка. Это немного не то, что мы хотели. Чтобы ограничить "жадность" регулярных выражений, нужно использовать символ ?:

```
>>> p = re.compile(r"<i>.*?</i>", re.S)
>>> p.findall(s)
['<i>Один</i>', '<i>Два</i>', '<i>Три</i>']
>>>
```

Теперь вместо одного большого соответствия у нас есть три меньших.

Мы только что рассмотрели основы синтаксиса регулярных выражений Perl. Дополнительную информацию вы можете получить по адресу:

http://www.boost.org/doc/libs/1_43_0/libs/regex/doc/html/boost_regex/syntax/perl_syntax.html

7.3. Методы match() и search()

Метод *match()* проверяет соответствие с началом строки. Формат метода следующий:

```
match(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Если соответствие найдено, то возвращается объект Match, в противном случае – *None*. Пример:

```
>>> p = re.compile('[0-9]+')
>>> print("Найдено" if p.match("s123") else "Не найдено")
Не найдено
```

```
>>> print("Найдено" if p.match("123s") else "Не найдено")
Найдено
>>>
```

Метод *search()* проверяет соответствие с любой частью строки:

```
search(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Пример:

```
>>> p = re.compile('[0-9]+')
>>> print("Найдено" if p.search("s123") else "Не найдено")
Найдено
>>> print("Найдено" if p.search("123s") else "Не найдено")
Найдено
```

Объект Match, возвращаемый методами match() и search() имеет следующие свойства и методы:

- re — ссылка на скомпилированный шаблон, указанный в методах match() и search() Через нее доступны следующие свойства и методы:
- groups – количество групп в шаблоне
- goupindex – словарь с названиями групп и их номерами
- string – значение параметра <Строка>
- pos – значение параметра <Начальная позиция>
- endpos – значение параметра <Конечная позиция>
- lastindex – номер последней группы или значение *None*
- lastgroup – название последней группы
- start([<Номер группы или название>]) – индекс начала фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае – соответствие с указанной группой. Если соответствия нет, то возвращается значение -1
- end([<Номер группы или название>]) – индекса конца фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае – с указанной группой. Если соответствия нет, то возвращается -1
- expand(<Шаблон>) – производит замену в строке согласно указанного шаблона

В качестве примера работы с регулярными выражениями рассмотрим код, проверяющий правильность введенного e-mail (лист. 7.1).

Листинг 7.1. Код, проверяющий правильность e-mail

```
import re

email = "mark@sales.example.com"

pattern = r"^([a-z0-9_.-]+)@(([a-z0-9-]+\.)+[a-z]{2,6})$"
p = re.compile(pattern, re.I | re.S)

m = p.search(email)

if not m:
   print("Не совпадает")
else:
   print("Совпадает")
```

7.4. Метод findall()

Метод *findall()* ищет все совпадения с шаблоном. Формат метода:

```
findall(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Если найдены соответствия, то возвращается список с фрагментами, в противном случае возвращается пустой список. Если внутри шаблона есть несколько групп, то каждый элемент списка будет кортежем, а не строкой. Рассмотрим пример:

```
>>> p = re.compile(r"[a-z]+")
>>> p.findall("abc, bca, 123, dsf")
['abc', 'bca', 'dsf']

>>> p.findall("1234, 12345, 123456")
[]
```

7.5. Метод sub()

Метод *sub()* используется для поиска всех совпадений в строке с шаблоном и для их замены указанным значением. Если совпадения не найдены, будет возвращена исходная строка. Синтаксис метода следующий:

```
sub(<новый фрагмент>, <строка для замены> [, <максимальное
количество замен>])
```

Внутри нового фрагмента можно использовать обратные ссылки \номер \gНомер и \gНазвание. Обратные ссылки предоставляют удобный способ идентификации повторяющегося символа или подстроки в строке. Например, если входная строка содержит несколько экземпляров произвольной подстроки, то можно найти первое вхождение с помощью группы записи, а затем использовать обратную ссылку для поиска последующих вхождений подстроки. Чаще всего используются ссылки типа \номер. Номер – это порядковое положение группы записи, определенное в шаблоне регулярного выражения. Например, \4 соответствует содержимому четвертой захватываемой группы.

В качестве первого параметра можно использовать ссылку на функцию. В функцию будет передан объект Match, соответствующий найденному фрагменту. Результат, возвращаемый этой функцией, служит фрагментом для замены. Для примера найдем все числа в строке и умножим их на 2 (лист. 7.2)

Листинг 7.2. Использование метода sub

```
import re
def mult(m):
   x = int(m.group(0))
   x *= 2
   return "{0}".format(x)

p = re.compile(r"[0-9]+")
# умножаем все числа на 2
print(p.sub(mult, "2, 3, 4, 5, 6, 7"))
# умножаем первые три числа
print(p.sub(mult, "2, 3, 4, 5, 6, 7", 3))
```

Результат:

```
4, 6, 8, 10, 12, 14
4, 6, 8, 5, 6, 7
```

Обратите внимание, что для вызова функции не нужно указывать фигурные скобки.

Далее будут рассмотрены некоторые примеры, демонстрирующие использование регулярных выражений на практике.

7.6. Различные практические примеры

7.6.1. РАЗДЕЛЕНИЕ СТРОК С ИСПОЛЬЗОВАНИЕМ РАЗДЕЛИТЕЛЕЙ

Представим, что у нас есть строка и нам ее нужно разделить на поля, используя разделители, например, пробелы. Чтобы задача была менее тривиальной, будем считать, что разделители и пробелы вокруг них противоречивы по всей строке.

В самых простых случаях можно использовать метод *split()* объекта строки. Но он не позволяет использовать несколько разделителей и учитывать возможное пространство вокруг разделителей. В нашем случае нужна большая гибкость, поэтому нам нужен использовать метод *split()* из модуля **re**:

```
>>> line = 'asdf fjdk; afed, fjek,asdf, foo'
>>> import re
>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

Метод *re.split()* полезен, потому что вы можете определить многократные образцы для разделителя. например, как показано в решении, разделитель может быть или запятой (,) или точкой с запятой (;) или пробелом, сопровождаемым любым количеством дополнительных пробельных символов. Каждый раз, когда найден образец, все соответствия ему становятся разделителем между любыми полями, лежащими по обе стороны от соответствия. Результат – список полей, как и в случае с *str.split()*.

При использовании *re.split()* нужно быть осторожным, образец регулярного выражения должен содержать группу захвата, заключенную в круглые скобки. Если используются группы захвата, то соответствующий текст также будет включен в результат. Например:

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf',
',', 'foo']
>>>
```

Получение символов разделителя может быть полезным в определенных контекстах. Например, возможно, вы нуждаетесь в символах разделителя, чтобы преобразовать выходную строку:

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', '']

>>> # Преобразуем строку с использованием тех же разделителей
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

Если вы не хотите получить символы разделителя в результате, но вам все еще нужны круглые скобки для группировки частей образца регулярного выражения, убедитесь, что вы используете группы не захвата, которая определяется как (?...). Например:

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

7.6.2. ИСПОЛЬЗОВАНИЕ МАСКИ ОБОЛОЧКИ

Этот пример показывает, как найти соответствие текста с использованием масок, которые часто используются при работе с Unix-оболочкой (например, *.py, Dat[0-9]*.csv и т.д.).

Для решения поставленной задачи мы используем модуль **fnmatch**, предоставляющий две функции – *fnmatch()* и *fnmatchcase()*,которые могут использоваться для решения поставленной задачи. Использование функций очень простое:

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('report.txt', '*.txt')
True
>>> fnmatch('rep.txt', '?oo.txt')
False
>>> fnmatch('Data45.csv', 'Data[0-9]*')
True
```

```
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'rep.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

Шаблоны *fnmatch()* используют те же правила, что и маски используемой файловой системы (которая зависит от используемой операционной системы в свою очередь). Например:

```
>>> # В OS X (Mac)
>>> fnmatch('rep.txt', '*.TXT')
False
>>> # В Windows
>>> fnmatch('rep.txt', '*.TXT')
True
>>>
```

Если важен регистр символов, то используйте *fnmatchcase()*. Метод *fnmatchcase()* различает строчные и прописные символы, которые вы предоставляете:

```
>>> fnmatchcase('rep.txt', '*.TXT')
False
>>>
```

Функциональность *fnmatch()* находится где-то между функциональностью простых строковых методов и полной мощью регулярных выражений. Если вам нужно обеспечить простой механизм для разрешения подстановочных символов в операциях обработки данных, часто – это разумное решение.

7.6.3. СОВПАДЕНИЕ ТЕКСТА В НАЧАЛЕ И КОНЦЕ СТРОКИ

Пример показывает, как произвести поиск в начале и конце строки определенного шаблона текста, например, расширение имени файла, название протокола и т.д.

Самый простой способ заключается в использовании методов *str.startswith()* и *str.endswith()*. Например:

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
```

```
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

Если вам нужно проверить на соответствие нескольким вариантам, просто предоставьте кортеж возможных вариантов функциям *startswith()* или *endswith()*:

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('.c', '.h'))
]
['foo.c', 'spam.c', 'spam.h'
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

7.6.4. ПОИСК ПО ШАБЛОНУ

Довольно часто на практике возникает необходимость найти текст, соответствующий определенному шаблону. Если искомый текст является простым литералом, чаще проще использовать базовые строковые методы, например, *str.find()*, *str.endswith()*, *str.startswith()* и т.п. Например:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> # Точное совпадение
>>> text == 'yeah'
False
>>> # Совпадение в начале или в конце
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False
>>> # Поиск позиции первого вхождения искомого текста в строку
>>> text.find('no')
10
>>>
```

Для более сложного соответствия используйте регулярные выражения и

модуль **re**. Чтобы проиллюстрировать основную механику использования

регулярных выражений, предположим, что вы хотите найти даты, определенные как цифры, например, "03/19/2021". Вот как это можно сделать:

```
>>> text1 = '03/19/2021'
>>> text2 = 'Mar 19, 2021'
>>>
>>> import re
>>> # Простое соответствие: \d+ означает соответствие 1 или
более цифрам
>>> if re.match(r'\d+/\d+/\d+', text1):
... print('да')
... else:
... print('нет')
...
да
>>> if re.match(r'\d+/\d+/\d+', text2):
... print('да')
... else:

... print('нет')
...
нет
>>>
```

Если нужно выполнить больше соответствий, используя тот же образец, имеет смысл скомпилировать образец регулярного выражения в объект. Например:

```
>>> datepat = re.compile(r'\d+/\d+/\d+')

>>> if datepat.match(text1):
... print('да')
... else:
... print('нет')
...
да
>>> if datepat.match(text2):
... print('да')
... else:
... print('нет')
...
нет
>>>
```

Метод *match()* всегда пытается найти совпадение в начале строки. Если вам нужно найти все вхождения, используйте метод *findall()*. Пример:

```
>>> text = 'Сегодня 19/12/2021. Завтра 20/12/2021.'
>>> datepat.findall(text)
['12/19/2021', '12/20/2021']
>>>
```

При определении регулярных выражений принято представлять группы захвата, заключая части образца в круглые скобки. Например:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

Группы захвата часто упрощают последующую обработку соответствую-

щего текста, потому что содержание каждой группы может быть извлечено индивидуально. Например:

```
>>> m = datepat.match('12/19/2021')
>>> m
<_sre.SRE_Match object at 0x1005d2750>
>>> # Извлекаем содержимое каждой группы
>>> m.group(0)
'12/19/2021'
>>> m.group(1)
'12'
>>> m.group(2)
'19'
>>> m.group(3)
'2021'
>>> m.groups()
('12', '19', '2021')
>>> month, day, year = m.groups()
>>>
>>> # Поиск всех соответствий
>>> text
'Сегодня 12/19/2021. Завтра 12/20/2021.'
>>> datepat.findall(text)
[('12', '19', '2021'), ('12', '20', '2021')]
>>> for month, day, year in datepat.findall(text):
... print('{}-{}-{}'.format(year, month, day))
...
```

```
2021-12-19
2021-12-20
>>>
```

Метод *findall()* ищет текст и находит все соответствия, возвращая их как список. Если вы хотите найти соответствия итеративно, используйте метод *finditer()*. Например:

```
>>> for m in datepat.finditer(text):
... print(m.groups())
...
('12', '19', '2021')
('12', '20', '2021')
>>>
```

Обсуждение теории регулярных выражений выходит за рамки этой книги. Однако этот пример иллюстрирует абсолютные основы использования модуля **re**. Сначала нужно скомпилировать образец, используя *re.compile()*, а затем использовать методы, такие как *match()*, *findall()* или *finditer()*.

При указании шаблонов, как правило, принято использовать обычные строки, такие как r'(\d+)/(\d+)/(\d+)'. В таких строках обратный слеш остается неинтерпретируемым, что может быть полезно в контексте регулярных выражений. Иначе вам придется использовать двойной обратный слеш, например, '(\\d+)/(\\d+)/(\\d+)'.

Знайте, что метод *match()* проверяет только начало строки. Возможно, это немного не то, что вы ожидаете. Например:

```
>>> m = datepat.match('11/27/2021abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2021'
>>>
```

Если вам нужно точное совпадение, убедитесь, что шаблон в конце содержит маркер $, например:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2021abcdef')
>>> datepat.match('11/27/2021')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

Наконец, если вы просто выполняете простой поиск текста, вы можете часто пропустить шаг компиляции и использовать функции уровня модуля в модуле **re** вместо этого. Например:

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('12', '19', '2021'), ('12', '20', '2021')]
>>>
```

Знайте, тем не менее, что если вы собрались произвести значительное соответствие или поиск по большому тексту, обычно имеет смысл сначала скомпилировать образец, чтобы использовать его снова и снова. Так вы получите значительный прирост производительность.

7.6.5. ПОИСК И ЗАМЕНА ТЕКСТА

В самых простых случаях используйте метод *str.replace()*. Например:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

Для более сложных образцов используйте функцию/метод *sub()* в модуле **re**. В качестве примера представим, что нужно перезаписать даты вроде "11/27/2021" так, чтобы они выглядели так: "2021-11-27".

Вот пример того, как это можно сделать:

```
>>> text = 'Сегодня 11/27/2021. Завтра 3/13/2021.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Сегодня 2021-11-27. Завтра 2021-3-13.'
>>>
```

Первый аргумент *sub()* – это шаблон, которой должен быть найден в тексте, а второй аргумент – это шаблон замены. Цифры с обратными слешами (например, \3) используются для группировки чисел в образце.

Если вы собираетесь выполнить повторные замены с тем же образцом, задумайтесь о его компиляции для лучшей производительности. Например:

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Сегодня 2021-12-19. Завтра 2021-12-20.'
>>>
```

Для более сложных замен можно использовать callback-функцию замены. Например:

```
>>> from calendar import month_abbr
>>> def change_date(m):
... mon_name = month_abbr[int(m.group(1))]
... return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Сегодня 19 Dec 2021. Завтра 20 Dec 2021.'
>>>
```

В качестве ввода в callback-функцию замены передается объект, возвращенный функциями *match()* или *find()*. Используйте метод .group() для извлечения определенных частей соответствия. Функция должна вернуть текст замены.

Если вы знаете, сколько замен было сделано в дополнение к получению текста замены, используйте вместо этого *re.subn()*. Например:

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Сегодня 2021-12-19. Завтра 2021-12-20.'
>>> n
2
>>>
```

Для выполнения нечувствительных к регистру операций над текстом вам нужно использовать модуль **re** и установить флаг re.IGNORECASE. Этот флаг нужно устанавливать отдельно для каждой операции над текстом. Например:

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

Последний пример показывает ограничение, что текст замены не будет соответствовать регистру в исходном тексте. Если вам нужно исправить это, вам, возможно, придется использовать функцию поддержки, например:

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
           return word.upper()
        elif text.islower():
           return word.lower()
        elif text[0].isupper():
           return word.capitalize()
        else:
           return word
return replace
```

Вот пример использования этой последней функции:

```
>>> re.sub('python', matchcase('snake'), text, flags=re.
IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

7.6.6. УДАЛЕНИЕ НЕЖЕЛАТЕЛЬНЫХ СИМВОЛОВ ИЗ СТРОКИ

Часто требуется удалить нежелательные символы, например, пробелы в начале и конце строки.

Метод *strip()* может быть использован для удаления символов в начале или конце строки. Функции *lstrip()* или *rstrip()* осуществляет обрезку символов слева или справа соответственно.

По умолчанию эти методы удаляют пробельные символы, но вы можете задать любые другие символы. Например:

```
>>> # Обрезаем пробельные символы
>>> s = ' привет мир \n'
>>> s.strip()
'привет мир'

>>> s.lstrip()
'привет мир \n'
>>> s.rstrip()
' привет мир'
>>>

>>> # Удаляем заданные символы
>>> t = '-----привет====='
>>> t.lstrip('-')
'привет====='
>>> t.strip('-=')
'привет'
>>>
```

Различные strip()-методы обычно используются при чтении и очистке данных для дальнейшей обработки. Например, вы можете использовать их, чтобы избавиться от пробелов, удалить кавычки и выполнить другие задачи. Знайте, что strip()-методы не применяются к любому тексту в середине строки. Например:

```
>>> s = ' привет \n'
>>> s = s.strip()
>>> s
'привет мир'
>>>
```

Если вам нужно сделать что-то во внутреннем пространстве, вам нужно использовать другую технику, например, использовать метод *replace()* или замену с использованием регулярных выражений. Например:

```
>>> s.replace(' ', '')
'приветмир'
>>> import re
>>> re.sub('\s+', ' ', s)
'привет мир'
>>>
```

Часто нужно объединить строковые операции разделения (*splitting*) с некоторым другим видом итеративной обработки, например, чтение строк данных из файла. Если это так, то вам пригодится использование генератора. Например:

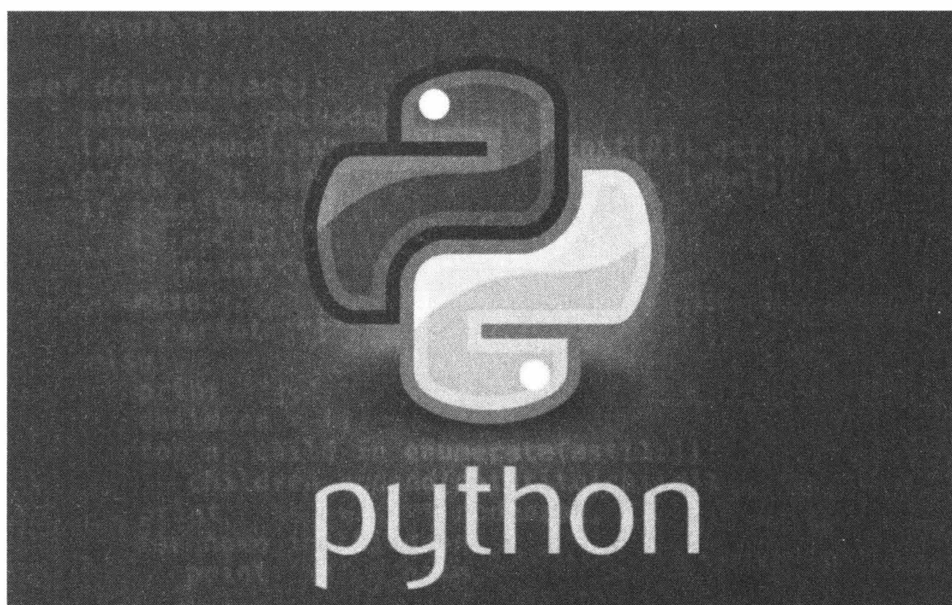
```
with open(filename) as f:
    lines = (line.strip() for line in f)
    for line in lines:

    ...
```

Здесь выражение *lines = (line.strip() for line in f)* действует как, своего рода, преобразование данных. Это эффективно, потому что оно, фактически, сначала не считывает данные в какой-либо вид временного списка. Мы просто создаем итератор, где все ко всем считанным строкам сразу применяется *strip()*.

ГЛАВА 8.

СПИСКИ



8.1. Что такое список?

Список – это нумерованный набор объектов. Каждый элемент набора содержит только ссылки на объект. Именно поэтому список может содержать объекты произвольного типа данных и иметь неограниченную степень вложенности.

Списки поддерживают обращение к элементу по индексу (нумерация элементов списка начинается с 0), получение среда, конкатенацию, повторение, а также проверку на вхождение.

Ранее было отмечено, что списки относятся к изменяемым типам данных (в отличие от кортежей). Это означает, что вы можете не только получить элемент по индексу, но и изменить его. Например:

```
>>> lst = [1, 2, 3, 4]
>>> lst[1]
2
>>> lst[1] = 7
>>> lst
[1, 7, 3, 4]
>>>
```

Создать список можно разными способами. Например, можно использовать функцию *list()*, которой нужно передать последовательность, по которой и будет создан список. Если вы ничего не передаете, будет создан пустой список:

```
>>> lst = list('Hello')
>>> lst
['H', 'e', 'l', 'l', 'o']
```

Можно указать все элементы списка в квадратных скобках, как уже было

показано. Обратите внимание, что элементы могут быть разных типов:

```
>>> lst = ["a", "b", 1]
>>> lst
['a', 'b', 1]
```

Третий способ заключается в поэлементном формировании списка с помощью метода **append**:

```
>>> lst = []
>>> lst.append(1)
>>> lst.append(2)
>>> lst.append(3); lst
[1, 2, 3]
>>>
```

В PHP можно использовать такую конструкцию для добавления элементов в список:

```
lst[] = новый_элемент
```

В Python ее использовать нельзя, вы получите сообщение об ошибке. Также нужно быть осторожнее со следующей конструкцией:

```
>>> a = b = ["a", "b"]
>>> a[0]
'a'
>>> b[0] = 1
>>> a[0]
1
>>>
```

Как видите, при создании списка сохраняется ссылка на объект, а не сам объект. Поэтому нам кажется, что мы якобы создали два списка, а на самом деле обоим переменным была присвоена ссылка, указывающая на объект. Напомню, что проверить, ссылаются ли переменные на один и тот же объект, можно с помощью оператора **is**, например (если оператор возвращает *True*, то переменные ссылаются на один и тот же объект):

```
>>> a is b
True
```

Если вам нужно создать вложенные списки, то это лучше делать с помощью метода *append()*, например:

```
>>> lst = []
>>> for i in range(3): lst.append([])
>>> lst[0].append(1)
>>> lst
[[1], [], []]
>>>
```

8.2. Операции над списками

Над списками можно выполнить множество операций. Начнем с доступа к элементу. Для этого используются квадратные скобки ([]), в которых указывается индекс элемента. Нумерация элементов списка начинается с 0. Примеры использования [] уже были показаны.

> ***Примечание.*** Поскольку нумерация элементов списка начинается с 0, то индекс последнего элемента будет меньше на единицу количества элементов.

Оператор присваивания можно использовать как для присваивания значения всему списку, так и отдельному элементу:

```
>>> lst = [1, 2, 3]
>>> lst[1] = 6
```

В Python 3 при позиционном присваивании перед одной из переменных слева от оператора = можно указать звездочку. В этой переменной

будет сохраняться список, состоящий из "лишних" элементов. Если таких элементов нет, то список будет пустым:

```
>>> a, b, *c = [1, 2, 3, 4]
```

Если вы попытаетесь получить доступ к несуществующему элементу, вы получите сообщение об ошибке *IndexError*:

```
>>> lst[4]
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    lst[4]
IndexError: list index out of range
```

В качестве индекса можно использовать отрицательные значения, в этом случае смещение будет отсчитываться с конца списка:

```
>>> lst = [1, 2, 3, 4, 5, 6, 7]
>>> lst[-2]
6
```

Кроме обращения к элементу по индексу списки поддерживают операцию извлечения среза, которая возвращает указанный фрагмент списка:

```
[<Начало>:<Конец>:<Шаг>]
```

Все три параметра являются необязательными. Если не указан первый параметр, то используется значение 0. Если не задан второй параметр, то считается, что нужно возвратить фрагмент до конца списка. Если не задан последний параметр, то используется значение 1.

Операция среза очень интересная и мощная. Вот, например, как можно вывести элементы списка в обратном порядке:

```
>>> lst[::-1]
[7, 6, 5, 4, 3, 2, 1]
```

Вот еще несколько примеров:

```
>>> lst[:-1]      # Без последнего элемента
```

```
[1, 2, 3, 4, 5, 6]
>>> lst[1:]       # Без первого элемента
[2, 3, 4, 5, 6, 7]
>>> lst[0:3]      # Первые три элемента
[1, 2, 3]
>>> lst[-1:]      # Последний элемент
[7]
```

Срез позволяет даже изменять элементы списка, например:

```
>>> lst[1:3] = [8, 9]
>>> lst
[1, 8, 9, 4, 5, 6, 7]
```

Только будьте осторожны с этой операцией!

Срез – это не единственная полезная операция над списком. Вы можете выполнить конкатенацию списков, используя оператор **+**:

```
>>> lst
[1, 8, 9, 4, 5, 6, 7]
>>> lst2 = [10, 11, 12]
>>> lst3 = lst + lst2; lst3
[1, 8, 9, 4, 5, 6, 7, 10, 11, 12]
```

Если нужно добавить элементы в текущий список, можно использовать оператор +=:

```
>>> lst
[1, 8, 9, 4, 5, 6, 7]
>>> lst += [10, 11, 13]; lst
[1, 8, 9, 4, 5, 6, 7, 10, 11, 13]
```

8.3. Многомерные списки

Любой элемент списка может содержать любой объект, в том числе и другой список, кортеж, словарь и т.д. Вот как можно создать список, состоящий из трех списков:

```
>>> lst = [[1, 2, 3], ['a','b','c'], [9, 8, 7]]
```

```
>>> lst
[[1, 2, 3], ['a', 'b', 'c'], [9, 8, 7]]
```

Чтобы добраться к элементам вложенного списка, нужно указывать два индекса, например:

```
>>> lst[1][2]
'c'
```

В свою очередь элементы вложенного списка также могут быть списками и т.д. Количество вложений не ограничено, поэтому у вас могут быть вот такие странные индексы:

```
lst[1][2][3][4]…
```

Если того не требует решаемая задача, я бы не советовал увлекаться вложенными списками – так вы сделаете программу сложнее, чем она могла бы быть.

8.4. Проход по элементам списка

Вот как можно перебрать все элементы списка:

```
>>> lst
[[1, 2, 3], ['a', 'b', 'c'], [9, 8, 7]]
>>> for i in lst: print(i, end=" ")
[1, 2, 3] ['a', 'b', 'c'] [9, 8, 7]
>>>
```

Также для перебора списка можно использовать функцию *range()*:

```
range([<Начало>,] <Конец> [, <Шаг>])
```

Пример:

```
>>> lst = [1, 2, 3, 4]
>>> for i in range(len(lst)): print(lst[i], end=" ")
1 2 3 4
```

Данный способ можно использовать не только для итерации по одномерным спискам, как вы бы могли подумать, например:

```
>>> lst = [[1, 2, 3], ['a','b','c'], [9, 8, 7]]
>>> for i in range(len(lst)): print(lst[i], end=" ")
[1, 2, 3] ['a', 'b', 'c'] [9, 8, 7]
```

В принципе, для перебора элементов списка можно использовать и цикл **while**, но обычно используется цикл **for**, что и было продемонстрировано.

8.5. Поиск элемента в списке

Определить, есть ли элемент в списке, можно с помощью оператора **in**, например:

```
>>> lst
[[1, 2, 3], ['a', 'b', 'c'], [9, 8, 7]]
>>> 2 in lst
False
>>> lst = [1, 2, 3, 4]
>>> 2 in lst
True
```

Как видите, данный способ не работает с многомерными списками, поэтому для поиска элемента в таких списках правильнее использовать перебор элементов. Хоть медленно, но зато работает.

Однако оператор **in** сообщает только, если ли элемент в списке, но он не сообщает его позицию. Для этого можно использовать метод **index**:

```
>>> lst.index(3)
2
```

Здесь видно, что элементу 3 соответствует индекс 2. Если указанного элемента нет в списке, вы получите ошибку *ValueError*:

```
>>> lst.index(7)
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    lst.index(7)
ValueError: 7 is not in list
```

Посчитать количество элементов с определенным значением позволяет метод *count()*:

```
>>> lst = [1, 2, 2, 3, 4]
>>> lst.count(2)
2
>>> lst.count(3)
1
>>> lst.count(7)
0
```

Данный метод можно также использовать в качестве основного метода поиска элемента – если количество равно 0, то и элемента в списке нет. Вам не нужно обрабатывать никакие исключения, просто анализируйте количество элементов и все.

Функции *min()* и *max()* позволяют найти минимальный и максимальный элемент списка соответственно:

```
>>> lst = [1, 2, 2, 3, 4]
>>> min(lst); max(lst)
1
4
```

8.6. Добавление и удаление элементов в списке

Для добавления и удаления элементов создано множество методов. Начнем с метода append(<объект>), позволяющего добавить элемент в конец списка:

```
>>> lst = [1, 2, 3, 4]
>>> lst.append(5)
>>> lst
[1, 2, 3, 4, 5]
```

Также добавить элементы в конец списка можно и с помощью оператора +=, например:

```
>>> lst += [6, 7]
>>> lst
[1, 2, 3, 4, 5, 6, 7]
```

Метод insert() вставляет объект в указанную позицию. Синтаксис следующий:

```
insert(<индекс>, <объект>)
```

Примеры:

```
>>> lst
[1, 2, 3, 4, 5, 6, 7]
>>> lst.insert(0, 0)      # Вставили ноль в начало списка
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7]
>>> lst.insert(8, 8)      # Вставили 8 в позицию 8
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Для удаления элементов можно использовать методы *pop()*, *remove()* и оператор **del**. Первый удаляет элемент с указанным индексом и возвращает его. Если индекс не указан, удаляется и возвращается последний элемент списка:

```
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> lst.pop(0)
0
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8]
>>> lst.pop()
8
>>> lst
[1, 2, 3, 4, 5, 6, 7]
```

Оператор **del** ничего не возвращает, а просто удаляет элемент. Например:

```
>>> del lst[6]
>>> lst
[1, 2, 3, 4, 5, 6]
>>>
```

Удалить элемент, содержащий определенное значение, можно с помощью метода *remove()*:

```
>>> lst.remove(5)
>>> lst
[1, 2, 3, 4, 6]
```

8.7. Перемешивание элементов и выбор случайного элемента

Функция *shuffle()* из модуля **random** используется для перемешивания списка случайным образом. Функция перемешивает сам список и ничего не возвращает. Пример:

```
>>> import random
>>> lst
[1, 2, 2, 3, 4]
>>> random.shuffle(lst)
>>> lst
[4, 2, 2, 1, 3]
>>>
```

Выбрать случайный элемент со списка можно с помощью функции *choice()* из того же модуля:

```
>>> random.choice(lst)
4
>>> random.choice(lst)
2
```

Изменить порядок следования элементов можно с помощью метода *reverse()*. В отличие от среза, данный метод работает с самим списком, а не с его копией:

```
>>> lst
[4, 2, 2, 1, 3]
>>> lst.reverse()
>>> lst
[3, 1, 2, 2, 4]
>>>
```

8.8. Сортировка списка

Для сортировки списка используется метод *sort()*, синтаксис которого следующий:

```
sort([key=None] [, reverse=False])
```

Оба параметра являются не обязательными. Метод изменяет текущий список и ничего не возвращает. Попробуем отсортировать список, используя параметры по умолчанию:

```
>>> lst = [2, 3, 7, 5, 6, 1, 4]
>>> lst.sort()
>>> lst
[1, 2, 3, 4, 5, 6, 7]
>>>
```

Для сортировки в обратном порядке укажите параметр reverse=True:

```
>>> lst.sort(reverse=True)
>>> lst
[7, 6, 5, 4, 3, 2, 1]
>>>
```

Иногда нужно не учитывать регистр символов, для этого нужно вызвать *sort()* так:

```
lst.sort(key=str.lower)
```

Помните, что метод *sort()* изменяет список, а в некоторых случаях этого не нужно делать. Для таких случаев предназначена функция sorted():

```
sorted(<Последовательность>[, key=None][, reverse=False])
```

Первый параметр – это последовательность, которую нужно отсортировать, остальные параметры – такие же, как у метода *sort()*. Данная функция возвращает отсортированный список и не изменяет исходный.

8.9. Преобразование списка в строку

Для преобразования списка в строку используется метод *join()*. Вызывать его нужно так:

```
<Строка> = <разделитель>.join(<последовательность>)
```

Пример:

```
>>> lst = ['h', 'e', 'l', 'l', 'o']
>>> s = "".join(lst)
>>> s
'hello'
```

Здесь в качестве разделителя мы используем пустую строку, поэтому, по сути, разделителя нет.

8.10. Вычисления с большими числовыми массивами

NumPy[1] – основа для огромного числа научных и технических библиотек Python. Но в то же время NumPy – один из самых больших и самых сложных в использовании модулей. Однако вы можете выполнить полезные вещи с NumPy, начиная с простых примеров и экспериментируя с ними.

Нужно сделать только одно примечание относительно использования NumPy. Относительно распространено использовать оператор `import numpy as np`, что и показано в далее. Это просто сокращает название модуля – так удобнее, если часто приходится обращаться к нему.

Представим, что вам нужно произвести вычисления с огромными числовыми наборами данных, например, с массивами или сетками (таблицами).

Для любых "тяжелых" вычислений с использованием массивов нужно использовать библиотеку NumPy. Основное назначение NumPy – то, что она предоставляет Python объект массива, который более эффективен и лучше подходит для математических вычислений, чем стандартный список Python. Вот небольшой пример, иллюстрирующий важные поведенческие различия между массивами NumPy и списками:

```
>>> # Списки Python
>>> a = [2, 2, 2, 2]
>>> b = [3, 3, 3, 3]
>>> a * 2
[2, 2, 2, 2, 2, 2, 2, 2]
>>> a + 10
Traceback (most recent call last):
  File "<pyshell#115>", line 1, in <module>
    a + 10
TypeError: can only concatenate list (not "int") to list
>>> a + b
[2, 2, 2, 2, 3, 3, 3, 3]
>>>

>>> # Массивы Numpy
>>> import numpy as np
```

1 http://www.numpy.org

```
>>> an = np.array([2, 2, 3, 3])
>>> bn = np.array([5, 5, 7, 7])
>>> an * 2
array([4, 4, 6, 6])
>>> an + 10
array([12, 12, 13, 13])
>>> an + bn
array([ 7, 7, 10, 12])
>>> an * bn
array([ 10, 10, 21, 31])
>>>
```

Как видите, основные математические операции с массивами ведут себя иначе. В частности, скалярные операции (например, an * 2 или bn + 10) применяются к массиву поэлементно (в случае с обычным списком нужно было писать цикл и добавлять в цикле 10 к каждому значению списка). Кроме того, математически операции, когда оба операнда являются массивами, применяются к каждому элементу и в результате создается новый массив.

Библиотека NumPy предоставляет набор "универсальных функций", которые также доступны для работы с массивами.

Данные функции представляют собой замену обычных функций, которые вы можете найти в модуле **math**. Например:

```
>>> np.sqrt(an)
array([1.41421356 , 1.41421356, 1.73205081, 1.73205081 ])
>>> np.cos(an)
array([-0.41614684, -0.41614684, -0.9899925 , -0.9899925])
>>>
```

Использование универсальных функций может быть в сотни раз быстрее, чем перебор массива в цикле поэлементно и выполнение необходимых операций над каждым элементом отдельно. Поэтому если вам нужно выполнить операции с использованием функций из модуля **math** над элементами массива, взгляните на аналогичные функции модуля np. Вы должны предпочесть их, если это возможно.

"За кулисами" массивы NumPy выделяются таким же способом, как и в C или Fortran. А именно они являются большими, непрерывными областями памяти, состоящие из однородного типа данных. Именно поэтому вы можете сделать массивы просто огромными, гораздо больше, чем вы себе можете представить. Например, если вы желаете сделать двумерную таблицу 10 000 x 10 000, состоящую из чисел с плавающей запятой, это не проблема.

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.],
       ...,
       [ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.]])
>>>
```

Все обычные операции все еще применяются ко всем элементам одновременно:

```
>>> grid += 100
>>> grid
array([[ 100., 100., 100., ..., 100., 100., 100.],
        [ 100., 100., 100., ..., 100., 100., 100.],
        [ 100., 100., 100., ..., 100., 100., 100.],
        ...,
        [ 100., 100., 100., ..., 100., 100., 100.],
        [ 100., 100., 100., ..., 100., 100., 100.],
        [ 100., 100., 100., ..., 100., 100., 100.]])
```

Один чрезвычайно известный аспект NumPy – способ, которым она расширяет список Python, индексирующий функциональность – особенно при работе с многомерными массивами. Чтобы проиллюстрировать это, сделайте простую двумерную матрицу и попробуйте выполнить некоторые эксперименты:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1, 2, 3, 4],
        [ 5, 6, 7, 8],
        [ 9, 10, 11, 12]])

>>> # Ряд 0
>>> a[1]
array([1, 2, 3, 4])

>>> # Колонка 1
>>> a[:,1]
array([ 2, 6, 10])

>>> # Выбираем фрагмент массива и изменяем его
```

```
>>> a[1:3, 1:3]
array([[ 6, 7],
       [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1, 2, 3, 4],
       [ 5, 16, 17, 8],
       [ 9, 20, 21, 12]])
```

8.11. Программа "Гараж"

Теория – это хорошо. Теперь продемонстрируем полученные знания на простом примере, который оформлен в виде листинга 8.1.

Листинг 8.1. Проход по списку автомобилей

```
cars = ["audi", "vw", "lexus"]
print("Наши машины: ")
for item in cars:
    print(item)
```

К списку можно применять функцию len():

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]
print("Всего ", len(cars), " машин(ы) в гараже")
```

Вывод программы:

```
Всего 5 машин(ы) в гараже
```

Оператор **in** можно использовать для поиска по списку. Рассмотрим небольшой пример:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]
car = input("Введите название авто: ")
if car in cars:
    print("У вас есть авто из списка!")
else:
    print("У вас нет машины из списка :(")
```

Вывод программы:

```
Введите название ато: vm
У вас нет машины из списка:(
```

```
Введите название авто: vw
У вас есть авто из списка!
```

Как уже было сказано, списки индексируются. Ничего нового нет. Индексация начинается с 0 (то есть первый элемент списка имеет индекс 0), поддерживаются положительные и отрицательные индексы.

Пример работы с индексами:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]

start = -len(cars)
end = len(cars)

for i in range(start,end,1):
    print("cars[", i, "] = ", cars[i])
```

Вывод программы:

```
cars[ -5 ] =  audi
cars[ -4 ] =  vw
cars[ -3 ] =  lexus
cars[ -2 ] =  gtr
cars[ -1 ] =  m5
cars[ 0 ] =  audi
cars[ 1 ] =  vw
cars[ 2 ] =  lexus
cars[ 3 ] =  gtr
cars[ 4 ] =  m5
```

Сначала мы получаем начало (-len()) и конец (len()) диапазона. Потом проходимся по списку и указываем в качестве индекса переменную **i**, которая изменяется от *start* до *end* с приростом в 1.

Из вывода видно, что первому элементу списка присвоен индекс 0. Также к нему можно обратиться по индексу -len(cars), который в нашем случае равен -5.

Теперь мы можем приступить к разработке самой программы "Гараж". Программа Гараж демонстрирует практически все операции над списком:

- Добавление и удаление элементов
- Вывод списка
- Сортировка списка

- Поиск элемента в списке

Листинг 8.2. Программа Гараж

```
cars = []

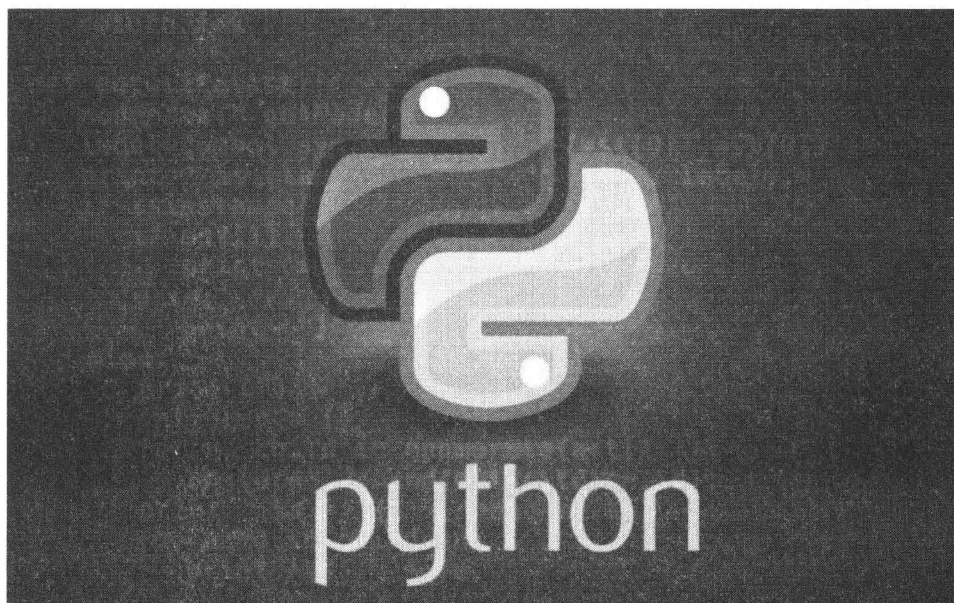
print("*" * 10, " Гараж v.0.0.1 ", "*" * 10)

responce = 1
while responce:
    print("""Выберите действие:
           1 - Добавить авто
           2 - Удалить авто
           3 - Вывести список авто
           4 - Поиск
           5 - Сортировка гаража
           0 - Выход""")
    responce = int(input(">> "))
    if responce == 1:
        car = input("Новое авто: ")
        cars.append(car)
    elif responce == 2:
        car = input("Удалить авто: ")
        cars.remove(car)
    elif responce == 3:
        print(cars)
    elif responce == 4:
        car = input("Поиск: ")
        if car in cars:
            print("Такая машина есть в гараже!")
        else:
            print("Нет такой машины в гараже!")
    elif responce == 5:
            cars.sort()
            print("Отсортировано!")
    else:
            print("Пока!")
```

Программа работает так. В цикле мы выводим подсказку-меню. Пользователь вводит свой выбор, программа выполняет действия в зависимости от введенного номера действия. Предполагается, что пользователь будет вводить только цифры от 0 до 5. Обработка некорректного ввода не добавлялась для упрощения кода.

ГЛАВА 9.

КОРТЕЖИ



Кортежи – это еще один из типов последовательностей. Но в отличие от строк, которые состоят только из символов, кортежи могут содержать элементы любой природы. В кортеже вы можете хранить, например, фамилии сотрудников, марки автомобилей, номера телефонов и т.д.

Самое интересное, что элементы кортежа не обязательно должны относиться к одному типу. При желании вы можете создать кортеж, содержащий, как строковые, так и числовые значения. Вообще кортеж может содержать все, что угодно – хоть звуковые файлы.

9.1. Понятие кортежа

В отличие от списка, кортежи относятся к неизменяемым типам данным. Это означает, что вы можете получить документ по индексу, но не можете его изменить. Небольшой пример, чтобы вы понимали, о чем речь:

```
>>> tup = (1, 2, 3, 4)
>>> tup[2]
3
>>> tup[2] = 5
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    tup[2] = 5
TypeError: 'tuple' object does not support item assignment
>>>
```

Как видите, при попытке присвоить новое значение элементу кортежа, мы получили сообщение о том, что кортежи не поддерживают присваивание значения элементу.

Кортежи, как и списки, являются упорядоченными последовательностями элементов. Во многом кортежи похожи на списки, разве что их только нельзя изменить. Грубо говоря, кортеж – это список, доступный "только для чтения".

9.2. Создание кортежей

Создать кортеж можно несколькими способами. Первый способ был уже нами рассмотрен – это перечисление элементов внутри круглых скобок через запятую, например:

```
tup = ()             # Создан пустой кортеж
tup = (1,)           # Кортеж из одного элемента
tup = (1, 2, 3)      # Кортеж из трех элементов
tup = (1, "str", 3) # Кортеж из трех элементов разного типа
```

Также можно использовать функцию *tuple()*, которая преобразует переданную ей последовательность в кортеж. Пример:

```
tup = tuple()              # Пустой кортеж
tup = tuple('Hello')       # Преобразуем строку в кортеж
tup = tuple([1, 2, 3])     # Преобразуем список в кортеж
```

Вообще-то кортеж формируют запятые, а не круглые скобки. Скажем, вы можете создать кортеж так:

```
>>> tup = 1, 2, 3
>>> tup
(1, 2, 3)
```

Позиция элемента в кортеже задается индексом. Нумерация элементов начинается с 0 (как и в случае со списком). Как и все последовательности, кортежи поддерживают обращение к элементу по индексу, получение среда, конкатенацию (+), проверку на вхождение (in) и повторение (*). Рассмотрим несколько примеров:

```
>>> tup = (1, 2, 3, 4, 5)
>>> tup[4]          # доступ по индексу
```

```
5
>>> tup[::-1]        # обратный порядок
(5, 4, 3, 2, 1)
>>> tup[2:3]         # срез
(3,)
>>> tup[1:3]         # еще срез
(2, 3)
>>> 7 in tup         # проверка на вхождение
False
>>> 2 in tup
True
>>> tup * 2          # повтор
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
>>> tup + (1, 2, 4) # конкатенация
(1, 2, 3, 4, 5, 1, 2, 4)
>>>
```

Примеры ранее были рассчитаны на интерактивную работу с IDLE. Теперь посмотрим, как можно создать кортеж в Python-файле:

```
cars = ("Nissan", "Toyota", "Lexus")
drivers = ()      # Создает пустой кортеж
```

Первая строка создает непустой кортеж, состоящий из трех элементов. Вторая строка создает пустой кортеж.

9.3. Методы кортежей

Кортежи поддерживают всего два метода:

```
index(<Значение>[, <Начало>[, <Конец>]])
count(<Значение>)
```

Первый метод возвращает индекс элемента с указанным значением. Если такого элемента нет в кортеже, то генерируется исключение ValueError. Если не заданы второй и третий параметры, то поиск будет производиться с начала кортежа.

```
>>> tup = (1, 2, 3, 4, 5, 2, 7)
>>> tup.index(2)
1
>>> tup.index(2, 2)
5
```

Второй метод подсчитывает количество элементов в кортеже с указанным значением:

```
>>> tup.count(2)
2
```

Чтобы не обрабатывать исключение ValueError, проверяйте сначала количество элементов методом *count()* – если оно отличное от 0, тогда вычисляйте позиции элементов методом *index()*.

Других методов у кортежей нет, но вы можете использовать функции, предназначенные для работы с последовательностями.

9.4. Перебор элементов кортежа

Вывести содержимое кортежа можно функцией *print()*:

```
print(cars)
```

Перебрать все элементы кортежа и что-то сделать с ними:

```
for item in cars:
    print(item)
```

9.5. Кортеж как условие

Кортеж можно использовать как условие, например:

```
if not cars:
  print("У вас нет машины!")
```

Пустой кортеж интерпретируется как ложное условие (*False*), а кортеж, содержащий хотя бы один элемент – как истинное. Поскольку пустой кортеж интерпретируется как *False*, то условие **not cars** оказывается истинным, поэтому программа выведет строку "У вас нет автомобиля".

9.6. Функция len() и оператор in

К кортежам может применяться функция len(), возвращающая число элементов кортежа:

```
print("Всего машин: ", len(cars))
```

Проверить существование элемента в кортеже можно так:

```
if "Nissan" in cars:
  print("У вас есть Nissan!")
```

9.7. Неизменность кортежей и слияния

О кортежах вам нужно знать еще две вещи. Первая – они, как и строки, неизменяемые:

```
>>> cars = ("nissan", "toyota")
>>> print(cars[0])
nissan
>>> cars[0] = "ford"
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    cars[0] = "ford"
TypeError: 'tuple' object does not support item assignment
>>>
```

То есть вы не можете присвоить другое значение элементу кортежа.

Вторая вещь – кортежи поддерживают слияния. Слияние кортежей еще называют сцеплением. Чтобы выполнить сцепление кортежей, нужно использовать оператор **+**.

9.8. Модуль itertools

Модуль **itertools** содержит функции, позволяющие генерировать различные последовательности на основе других последовательностей, производить фильтрацию элементов и др. Подключить модуль можно так:

```
import itertools
```

Далее можно использовать функции этого модуля. Начнем с функции *count()*, которая создает бесконечную нарастающую последовательность. Синтаксис:

```
count([start=0][, step=1])
```

Первый параметр задает начальное значение, а второй – шаг. Пример:

```
>>> import itertools as it
>>> for i in it.count():
   if i > 15: break
   print(i, end=" ")

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Просто так вызвать *count()* вы не можете, поскольку функция создает бесконечную последовательность и вы получаете бесконечный цикл.

Функция *repeat()* возвращает объект указанное количество раз. Функции передаются два параметра – объект и количество повторений. Если количество повторений не указано, то объект будет возвращаться бесконечно. Пример:

```
>>> list(it.repeat('*', 10))          # Список из '*'
['*', '*', '*', '*', '*', '*', '*', '*', '*', '*']

# Комбинация функций zip() и repeat()
>>> list(zip(it.repeat(3), "abc"))
[(3, 'a'), (3, 'b'), (3, 'c')]
>>>
```

В предыдущем примере для создания комбинаций мы использовали функцию zip(), которая не является частью **itertools**, но в самом модуле itertools есть подобная функциональность:

```
>>> list(it.combinations('abc', 2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

Функция *combinations_with_replacement(iterable, r)* создает комбинации длиной **r** из **iterable** с повторяющимися элементами. Пример:

```
>>> list(it.combinations_with_replacement ([1,2,3], 2))
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Для создания перестановок используется функция *permutations()*:

```
>>> list(it.permutations('abc', 2))
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'),
('c', 'b')]
```

Функция *cycle(iterable)* – возвращает по одному значению из последовательности, повторенной бесконечное число раз.

Функция *chain(*iterables)* – возвращает по одному элементу из первого итератора, потом из второго, до тех пор, пока итераторы не кончатся.

Функция *dropwhile(func, iterable)* возвращает элементы *iterable*, начиная с первого, для которого func вернула ложь. Пример:

```
>>> list(it.dropwhile(lambda x: x < 5, [1,4,6,4,1]))
[6, 4, 1]
>>>
```

Функция *tee(iterable, n=2)* создает кортеж из **n** итераторов:

```
it.tee([1,2,3,4], 2)
(<itertools._tee object at 0x04756558>, <itertools._tee object
at 0x04756508>)
>>>
```

Модуль **itertools** очень удобен, когда нужно создать перестановку, комбинацию. В этом случае не нужно изобретать колесо, а использовать функции модуля itertools.

9.9. Распаковка кортежа в отдельные переменные

Представим, что у нас есть кортеж из N элементов, который вы хотите "распаковать" в набор из N переменных.

Любая последовательность может быть распакована в переменные с использованием простой операции присваивания. Требование только одно: чтобы число переменных соответствовало числу элементов в структуре. Например:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'Den', 50, 91.1, (2022, 12, 21) ]
>>> name, shares, price, date = data
>>> name
'Den'
>>> date
(2022, 12, 21)
>>> name, shares, price, (year, mon, day) = data
>>> name
'Den'
>>> year
2022
>>> mon
12
>>> day
21
>>>
```

Если будет несоответствие в числе элементов, то вы получите ошибку. Например:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
  >>>
```

Фактически, распаковка работает с любым объектом, который является итерируемым, а не только с кортежами или списками. К таким объектам относятся строки, файлы, итераторы и генераторы. Например:

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
```

```
'e'
>>> e
'o'
>>>
```

При распаковке иногда бывает нужно отбросить определенные значения.

У Python нет специального синтаксиса для этого, но вы можете просто указать имя переменной для значений, которые нужно отбросить. Например:

```
>>> data = [ 'Den', 50, 91.1, (2022, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

Однако убедитесь, что имя переменной, которое вы выбираете, не используется для чего-то еще.

Ситуация усложняется, когда вам нужно распаковать N элементов из итерируемого объекта, который может быть длиннее, чем N, что вызывает исключение "*too many values to unpack*".

Для решения этой задачи может использоваться "звездочка". Например, предположим, что в конце семестра вы решаете отбросить первые и последние классы домашней работы и выполнить только их среднюю часть. Если классов только четыре, то можно распаковать все четыре, но, что если 24? Тогда все упрощает "звездочка":

```
def drop_first_last(grades):
  first, *middle, last = grades
  return avg(middle)
```

Рассмотрим и другой вариант использования. Предположим, что у вас есть записи, состоящие из имени пользователя и адреса электронной почты, сопровождаемые произвольным числом телефонных номеров. Вы можете распаковать эти записи так:

```
>>> record = ('Mark', 'mark@nit.center', '25-333-26', '888-12-11')
>>> name, email, *phone_numbers = user_record
>>> name
'Mark'
```

```
>>> email
'mark@nit.center'
>>> phone_numbers
['25-333-26', ''888-12-11']
>>>
```

Стоит отметить, что переменная phone_numbers всегда будет списком, независимо от того, сколько телефонных номеров распаковано (даже если ни один). Таким образом, любой код, использующий phone_numbers, должен всегда считать ее списком или хотя бы производить дополнительную проверку типа.

Переменная со звездочкой может также быть первой в списке. Например, скажем, что у вас есть последовательность значений, представляющая объемы продаж вашей компании за последние 8 кварталов.

Если вы хотите видеть, как самый последний квартал складывается в средних по первым семи кварталам, вы можете выполнить подобный код:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

А вот как выглядит эта операция из интерпретатора Python:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

Расширенная распаковка итерируемого объекта нестандартна для распа-

ковки итерируемых объектов неизвестной или произвольной длины. Часто у таких объектов есть некоторый известный компонент или образец в их конструкции (например, "все, что после элемента 1 считать телефонным номером") и распаковка со звездочкой позволяет разработчику усиливать те образцы, чтобы получить соответствующие элементы в итерируемом объекте.

Стоит отметить, что *-синтаксис может быть особенно полезным при итерации по последовательности кортежей переменной длины. Например, возможно у нас есть последовательность теговых кортежей:

```
records = [
```

```
('foo', 1, 2),
('bar', 'hello'),
('foo', 3, 4),
]

def do_foo(x, y):
  print('foo', x, y)

def do_bar(s):
  print('bar', s)

for tag, *args in records:
  if tag == 'foo':
    do_foo(*args)
  elif tag == 'bar':
    do_bar(*args)
```

Распаковка со звездочкой может также быть полезной, когда объединена с определенными видами операций обработки строк, например, при разделении строки (*splitting*). Например:

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

Иногда нужно распаковать значения и отбросить их. Вы не можете указать пустое место с * при распаковке, но вы можете использовать звездочку вместе с переменной _. Например:

```
>>> record = ('Den', 50, 123.45, (17, 03, 2021))
>>> name, *_, (*_, year) = record
>>> name
'Den'
>>> year
2021
>>>
```

Есть определенная схожесть между звездообразными функциями распаковки и обработки списков различных функциональных языков. Например, если у вас есть список, вы можете легко разделить его на компоненты головы и хвоста. Например:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

Можно было предположить написание функций, выполняющих такое разделение, в виде некоторого умного рекурсивного алгоритма. Например:

```
>>> def sum(items):
... head, *tail = items
... return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

Однако знайте, что рекурсия действительно не сильная функция Python из-

за свойственного ей предела. Таким образом, этот последний пример приведен только из академического любопытства, на практике вы вряд ли будете использовать рекурсию в Python.

9.10. Списки vs кортежи

Мы только что рассмотрели, как работать с кортежами в Python. Возникает закономерный вопрос – когда лучше использовать списки, а когда – кортежи? Понятно, что списки – лучше кортежей, поскольку можно изменять элементы списка.

Но не спешите отказываться от кортежей. У них есть следующие преимущества:

- Кортежи работают быстрее. Система знает, что кортеж не изменится, поэтому его можно сохранит так, что операции с его элементами будут выполняться быстрее, чем с элементами списка. В небольших программа

эта разница в скорости никак не проявит себя. Но при работе с большими последовательностями разница будет ощутимой.

- Неизменяемость кортежей позволяет использовать их как константы.
- Кортежи можно использовать в отдельных структурах данных, от которых Python требует неизменимых значений.
- Кортежи потребляют меньше памяти. Рассмотрим пример:

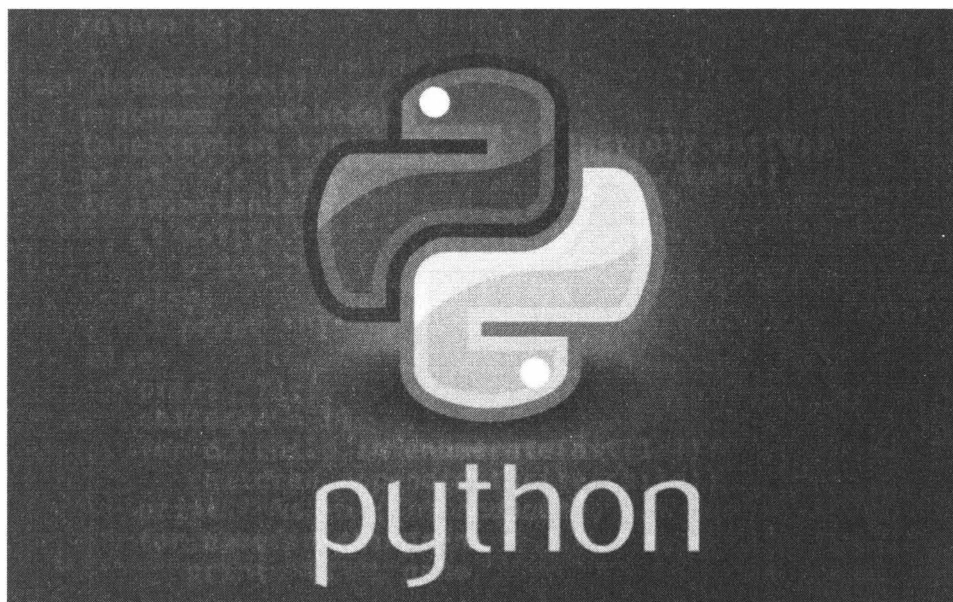
```
>>> a = (1, 2, 3, 4, 5, 6)
>>> b = [1, 2, 3, 4, 5, 6]
>>> a.__sizeof__()
36
>>> b.__sizeof__()
44
```

- Кортежи можно использовать в качестве ключей словаря:

```
>>> d = {(1, 1, 1) : 1}
>>> d
{(1, 1, 1): 1}
>>> d = {[1, 1, 1] : 1}
Traceback (most recent call last):
  File "", line 1, in
    d = {[1, 1, 1] : 1}
TypeError: unhashable type: 'list'
```

ГЛАВА 10.

МНОЖЕСТВА И СЛОВАРИ



10.1. Понятие словаря

Словарь – это набор объектов, доступ к которым осуществляется не по индексу, а по ключу (аналог ассоциативных массивов в PHP). Словари могут содержать данные разных типов и иметь неограниченную степень вложенности.

Элементы в словарях находятся в произвольном порядке. Для доступа к элементу нужно использовать ключ, нет никакого способа обратиться к элементу в порядке добавления.

Словари, как тип данных, относятся не к последовательностям, а к отображениям. Именно поэтому операции, которые были применимы к последовательностям (конкатенация, повторение, срез и т.д.), к словарям не применимы.

Существует несколько способов создания словаря. Первый способ – это использование функции *dict()*.

```
dict(<Ключ1>=<Значение1>[,…, <КлючN>=<ЗначениеN>]
dict(<Список кортежей с двумя элементами - Ключ и Значение>)
dict(<Список списков с двумя элементами - Ключ и Значение>)
```

Рассмотрим несколько примеров:

```
>>> d = dict()
>>> d = dict(name='Иван', surname='Иванов'); d
{'name': 'Иван', 'surname': 'Иванов'}
>>> d = dict({"name": "Иван", "surname": "Иванов"}); d
{'name': 'Иван', 'surname': 'Иванов'}
>>> d = dict([["name", "Иван"], ["surname", "Иванов"]]); d
{'name': 'Иван', 'surname': 'Иванов'}
>>> d = dict([("name", "Иван"), ("surname", "Иванов")]); d
{'name': 'Иван', 'surname': 'Иванов'}
>>>
```

Первый оператор создает пустой словарь. Второй – создает словарь по парам Ключ=Значение. Третий оператор – создает словарь по словарю, да и в качестве параметров функции dict() мы передали уже готовый словарь.

Четвертый оператор создает словарь по списку списков, а пятый – по списку кортежей. Как видите, существуют различные способы создания словарей, и вы можете выбрать тот, который вам больше нравится.

В создании словаря может участвовать и функция zip(). Она может объединить два списка в список кортежей, а затем созданный нею список мы можем передать в функцию dict(). Например:

```
>>> keys = ("name", "surname")
>>> values = ("Иван", "Иванов")
>>> list(zip(keys, values))
[('name', 'Иван'), ('surname', 'Иванов')]
>>> kv = list(zip(keys, values))
>>> d = dict(kv); d
{'name': 'Иван', 'surname': 'Иванов'}
>>>
```

У нас есть два списка – keys (ключи) и values (значения). Мы комбинируем их функцией **zip** и создаем общий список **kv**, который мы потом передаем в функцию **dict** и получаем такой же словарь, как и раньше.

Также создать словарь можно, заполнив его поэлементно, например:

```
>>> d = {}
>>> d["name"] = "Иван"
>>> d["surname"] = "Иванов"
>>> d
{'name': 'Иван', 'surname': 'Иванов'}
>>>
```

Если вам удобно, вы можете указать все элементы словаря в фигурных скобках:

```
>>> d = {}
>>> d = {"name": "Иван", "surname": "Иванов"}; d
{'name': 'Иван', 'surname': 'Иванов'}
>>>
```

При создании словаря нужно помнить, что в переменную сохраняется не

сам словарь, а только ссылка на него, что нужно учитывать при групповом присваивании. Если вам нужно скопировать словарь, то вам нужно использовать не оператор присваивания, а метод *copy()*. Рассмотрим пример:

```
>>> d = {"name": "Иван", "surname": "Иванов"}; d
{'name': 'Иван', 'surname': 'Иванов'}
>>> d2 = d
>>> d2 is d
True
>>> d2 = d.copy()
>>> d2 is d
False
>>>
```

Если присвоить **d** переменной d2, то оператор **is** сообщит, что обе переменные ссылаются на один и тот же объект в памяти (*True*). Если же скопировать словарь через метод *copy()*, то будет создана независимая копия в памяти (оператор **is** вернет *False*). Однако метод *copy()* делает только поверхностную копию словаря, для создания полной копии лучше использовать функцию *deepcopy()*:

```
>>> import copy
>>> d2 = copy.deepcopy(d); d2
{'name': 'Иван', 'surname': 'Иванов'}
>>> d2 is d
False
```

10.2. Различные операции над словарями

10.2.1. ДОСТУП К ЭЛЕМЕНТУ

Начнем с доступа к элементу. Доступ осуществляется по ключу:

```
>>> d["name"]
'Иван'
```

При обращении к несуществующему элементу будет сгенерировано исключение:

```
>>> d["lastname"]
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    d["lastname"]
KeyError: 'lastname'
```

Проверить наличие ключа можно с помощью оператора **in**:

```
>>> "surname" in d
True
>>> "lastname" in d
False
>>>
```

Узнать, сколько ключей есть в словаре можно с помощью функции *len()*:

```
>>> len(d2)
2
>>>
```

10.2.2. ДОБАВЛЕНИЕ И УДАЛЕНИЕ ЭЛЕМЕНТОВ СЛОВАРЯ

Добавить элемент в словарь можно так:

```
>>> d
{'name': 'Иван', 'surname': 'Иванов'}
>>> d["lastname"] = "Иванов"
>>> d
{'lastname': 'Иванов', 'name': 'Иван', 'surname': 'Иванов'}
```

Если ключ есть в словаре, то ему будем присвоено новое значение. Если ключа нет, то он будет добавлен в словарь.

Удалить ключ из словаря можно с помощью оператора **del**:

```
>>> d
{'lastname': 'Иванов', 'name': 'Иван', 'surname': 'Иванов'}
>>> del d["lastname"]; d
{'name': 'Иван', 'surname': 'Иванов'}
```

10.2.3. ПЕРЕБОР ЭЛЕМЕНТОВ СЛОВАРЯ

Перебрать все элементы словаря можно так:

```
>>> for key in d.keys():
   print("({0} => {1})".format(key, d[key]), end=" ")

(name => Иван) (surname => Иванов)
```

10.2.4. СОРТИРОВКА СЛОВАРЯ

Словарь – это неупорядоченная структура данных. Поэтому при выводе словаря его ключи выводятся в произвольном порядке. Вы же можете отсортировать словарь по ключам. Для этого нужно получить сначала список всех ключей, а затем использовать метод *sort()*:

```
>>> keys = list(d.keys())
>>> keys.sort()
>>> for key in keys:
  print("({0}=> {1})".format(key, d[key]), end=" ")

(name=> Иван) (surname=> Иванов)
```

Данный пример не очень удачен, поскольку и до сортировки ключи в словаре находились в отсортированном порядке (так получилось), но если до-

бавить в словарь новый элемент и повторить пример, все будет работать как нужно:

```
>>> d["lastname"]="Иванов"
>>> d["zip"]="109011"
>>> d
{'zip': '109011', 'lastname': 'Иванов', 'name': 'Иван',
'surname': 'Иванов'}
>>> keys = list(d.keys())
>>> keys.sort()
>>> for key in keys:
   print("({0}=> {1})".format(key, d[key]), end=" ")

(lastname=> Иванов) (name=> Иван) (surname=> Иванов) (zip=> 109011)
```

10.2.5. МЕТОДЫ KEYS(), VALUES() И НЕКОТОРЫЕ ДРУГИЕ

Метод *keys()*, как вы уже заметили, возвращает объект *dict_keys*, содержащий все ключи словаря. Данный объект поддерживает итерации, а также операции над множествами.

Аналогично, метод *values()* возвращает объект *dict_values*, содержащий все значения словаря. Данный объект также поддерживает итерации. Пример:

```
>>> values = d.values()
>>> list(values)
['109011', 'Иванов', 'Иван', 'Иванов']
```

Также у словарей есть много дополнительных и бессмысленных методов. Например, метод *get()* возвращает значение элемента, но его и так можно получить:

```
>>> d.get("lastname")
'Иванов'
>>> d["lastname"]
'Иванов'
```

Особого смысла в этом методе нет, как и в методе *clear()*, который очищает словарь. А вот метод *pop()* может пригодиться. Он удаляет элемент и возвращает его значение:

```
>>> d.pop("lastname")
'Иванов'
>>> d
{'zip': '109011', 'name': 'Иван', 'surname': 'Иванов'}
```

10.2.6. ПРОГРАММА DICT

Продемонстрируем полученные знания на примере простой программы-словаря. В этой программе мы будем активно использовать оператор **in**, чтобы выяснить, есть ли слово в словаре или нет:

```
if "bus" in dict:
    print(dict["bus"])
else:
   print("Слова нет в словаре!")
```

Поскольку при обращении к несуществующему элементу словаря генерируется ошибка, то перед обращением неплохо бы проверить его наличие с помощью оператора **in**. Напишем простейшую программу поиска по словарю.

Листинг 10.1. Словарь v0.1

```
dict = {
    "apple" : "яблоко",
    "bold" : "жирный",
    "bus" : "автобус",
    "cat" : "кошка",
    "car" : "машина"}

print("=" * 15, "Dict", "=" * 15)

word = ""
while word != "q":
    word = input("Введите слово или q для выхода: ")
    if word != "q":
```

```
        if word in dict:
            print(dict[word])
        else:
            print("Не найдено")
```

Программа осуществляет поиск по словарю. Посмотрим, как она организована. Сначала мы определяем переменную **word**. Цикл **while** будет работать, пока эта переменная не равна "q".

В цикле пользователю предлагается ввести слово. Если слово не равно "q", то мы начинаем поиск по словарю. Если слово найдено, мы выводим его значение, если нет – то строку "Не найдено".

Если пользователь введет "q", то в цикле мы ничего не делаем, а при следующей итерации цикл будет прекращен.

Продолжим разработку нашего **Словаря**. Попробуем модифицировать исходную программу так, чтобы она поддерживала добавление и удаление элементов словаря, а также некоторые другие возможности.

Листинг 10.2. Словарь v 0.1

```
# Словарь заполнен по умолчанию
dict = {
    "apple" : "яблоко",
    "bold" : "жирный",
    "bus" : "автобус",
    "cat" : "кошка",
    "car" : "машина"}

print("=" * 15, "Dict v 0.2", "=" * 15)

# Справка. Будет выведена по команде h
help_message = """
s - Поиск
a - Добавить новое слово
r - Удалить слово
k - Показать все слова
d - Показать весь словарь
h - Справка
q - Выход
"""

choice = ""
while choice != "q":
    choice = input("(h - help)>> ")
```

```
    if choice == "s":
        word = input("Введите слово: ")
        res = dict.get(word, "Не найдено!")
        print(res)
    elif choice == "a":
        word = input("Введите слово: ")
        value = input("Введите перевод: ")
        dict[word] = value
        print("Слово добавлено!")
    elif choice == "r":
        word = input("Введите слово: ")
        del dict[word]
        print("Слово удалено")
    elif choice == "k":
        print(dict.keys())
    elif choice == "d":
        for word in dict:
            print(word, ": ", dict[word])
    elif choice == "h":
        print(help_message)
    elif choice == "q":
        continue;
    else:
        print("Нераспознанная команда. Введите h для справки")
```

Основной цикл программы:

```
choice = ""
while choice != "q":
    choice = input("(h - help)>> ")
```

При каждой итерации мы выводим подсказку (h – справка)>> и читаем ввод пользователя. Справка, а именно доступные команды отображаются по команде **h**.

Поиск слова в словаре мы производим с помощью метода *get()*:

```
    if choice == "s":
        word = input("Введите слово: ")
        res = dict.get(word, "Не найдено!")
        print(res)
```

Добавление осуществляется так:

```
    elif choice == "a":
        word = input("Введите слово: ")
```

```
        value = input("Введите перевод: ")
        dict[word] = value
        print("Слово добавлено!")
```

Вывод словаря осуществляется в удобном для человека формате:

```
    elif choice == "d":
        for word in dict:
            print(word, ": ", dict[word])
```

А вот вывод всех слов подойдет разве что для отладки. При желании вы можете модифицировать код, чтобы он выводил список слов в удобном для человека формате:

```
    elif choice == "k":
        print(dict.keys())
```

Вывод будет таким:

```
(h - help)>> k
dict_keys(['bus', 'apple', 'cat', 'bold', 'phone', 'car'])
```

При вводе неизвестной команды программа выводит соответствующее сообщение, а при вводе **q** происходит выход из программы:

```
 elif choice == "q":
  continue;
 else:
  print("Нераспознанная команда. Введите h для справки ")
```

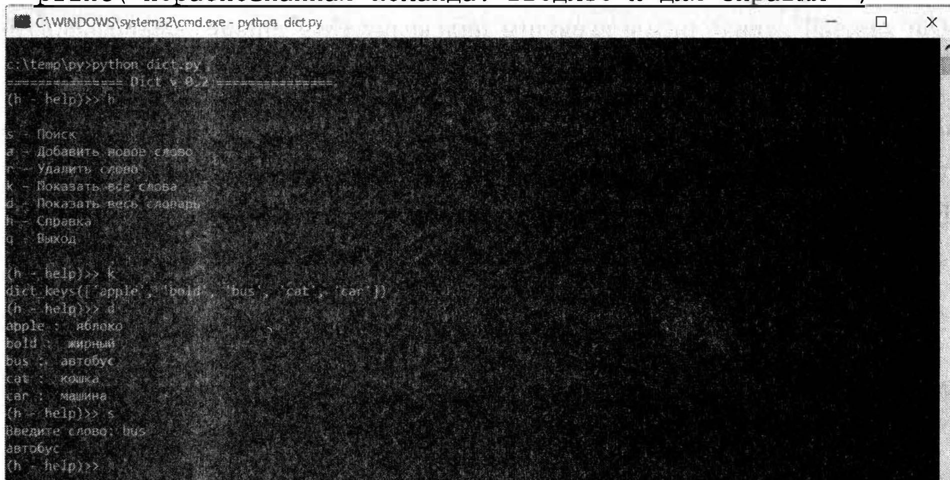


Рис. 10.1. Программа в действии

При вводе "q" мы вызываем **break**, чем инициируем переход на следующую итерацию. Далее в цикле **while** будет проверено значение и произведен выход из цикла. В принципе, можно было бы использовать **break**, чтобы сразу прервать работу цикла.

10.3. Понятие множества

Множество – это неупорядоченный набор уникальных элементов. Вы можете проверить, входит ли тот или иной элемент в множество. Множество не может содержать два одинаковых элемента, все элементы множества уникальны.

Создать множество можно с помощью функции *set()*:

```
>>> s = set()
>>> s
set()
```

Мы только что создали пустое множество. Однако функция *set()* может преобразовать во множество другие типы данных – строки, кортежи, списки. При преобразовании других типов данных помните, что во множестве останутся только уникальные элементы:

```
>>> s = set("Hello"); s                          # Строка
{'l', 'o', 'e', 'H'}
>>> set([1, 2, 3, 4, 5, 4])                      # Список
{1, 2, 3, 4, 5}
>>> set((1, 2, 3, 3, 4, 5))                      # Кортеж
{1, 2, 3, 4, 5}
```

10.4. Операции над множеством

Вот как можно перебрать элементы множества:

```
>>> for i in s: print(i, end=" ")

l o e H
```

Узнать количество элементов во множестве можно с помощью функции *len()*:

```
>>> len(s)
4
```

Но самое главное – не это. Прелесть множества в специальных операторах, предназначенных специально для множеств. Оператор | означает объединение множеств:

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([3, 4, 5])
>>> s3 = s1 | s2; s3
{1, 2, 3, 4, 5}
>>>
```

Обратите внимание, что при объединении множеств, в созданное множество попадают лишь уникальные элементы, что и продемонстрировано в этом примере.

С помощью | можно добавить в одно множество элементы другого множества:

```
>>> s1 |= s2; s1
{1, 2, 3, 4, 5}
```

Оператор – означает разницу множеств:

```
>>> s1
{1, 2, 3, 4, 5}
>>> s2
{3, 4, 5}
>>> s1 - s2
{1, 2}
```

Оператор s1 -= s2 удалит из множества s1 элементы, которые существуют и во множестве s1, и во множестве s2.

Оператор & – это пересечение множеств. Результат пересечения – это элементы, которые есть в обоих множествах:

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([3, 4, 5])
>>> s1 & s2
{3}
```

Оператор ^ возвращает элементы обоих множеств, исключая одинаковые элементы:

```
>>> s1 ^ s2
{1, 2, 4, 5}
```

Оператор **in** обеспечивает проверку наличия элемента во множестве:

```
>>> s1
{1, 2, 3}
>>> 3 in s1
True
>>> 4 in s1
False
```

Оператор **==** обеспечивает проверку на равенство множеств:

```
>>> s1 == s2
False
>>>
```

Оператор s1 <= s2 проверяет, входят ли все элементы множества s1 во множество s2:

```
>>> s1
{1, 2, 3}
>>> s2
{3, 4, 5}
>>> s1 <= s2
False
```

Оператор s1 < s2 проверяет, входят ли все элементы s1 во множество s2, но при этом сами множества не должны быть равны. Аналогично, есть операторы >= и >.

10.5. Методы множеств

Множества поддерживают следующие методы:

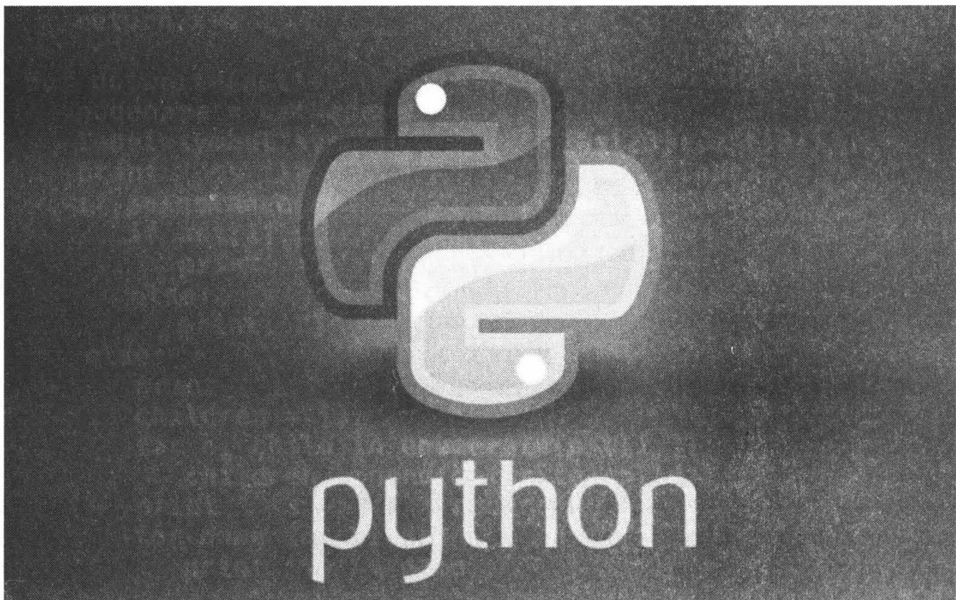
- add(<Элемент>) – добавляет <Элемент> во множество
- remove(<Элемент>) – удаляет <Элемент> из множества
- discard(<Элемент>) – удаляет указанный элемент из множества
- pop() – удаляет произвольный элемент и возвращает его
- clear() – очищает множество

Методы *remove()* и *discard()* отличаются тем, что если указанный элемент отсутствует во множестве, то в первом случае будет возвращена ошибка, а во втором никаких сообщений не будет:

```
>>> s1
{1, 2, 3}
>>> s1.add(4)
>>> s1.remove(4); s1
{1, 2, 3}
>>> s1.remove(4)
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    s1.remove(4)
KeyError: 4
>>> s1.discard(4)
```

ГЛАВА 11.

ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ



11.1. Объявление функции

Прежде всего, нужно разобраться, что такое **функция**. В любом языке программирования имеются подпрограммы, которые служат для экономии кода программы и удобства программиста. В Python подпрограммы называются функциями. Функции, как в любом другом языке программирования, могут возвращать значения. Заметьте: могут возвращать, а могут и нет – все зависит от того, какую функцию вы разрабатываете. Вы можете написать функцию, добавляющую переданные ей аргументы в текстовый файл или в базу данных, но не выводящую никакого текста и не возвращающую никаких значений.

Функциям в Python можно передавать параметры (аргументы). Примечательно, что Python (как и некоторые другие языки программирования) позволяет создавать функции с произвольным числом параметров и функции с параметрами по умолчанию, что облегчает использование функций

Функция может возвращать любой тип данных с помощью оператора **return**, но, как уже было отмечено, использование этого оператора необязательно. Напишем первую нашу функцию, чтобы продемонстрировать некоторые особенности функций в Python.

Объявление функции начинается со служебного слова *def*. Функцию можно объявить в любом месте сценария (хотя рекомендуется объявлять функции в начале сценария или вообще вывести функции в отдельный файл, который вы будете подключать с помощью инструкции *import*), но до первого ее вызова. Формат объявления функции следующий:

```
def <Имя_функции>([Параметры]):
   <Тело функции>
   [return <Значение>]
```

Имя функции – это уникальный идентификатор, состоящий из латинских букв, цифр и знаков подчеркивания. Параметры функции указываются в круглых скобках. Функция может иметь переменное число параметров, об этом мы поговорим позже.

Функция может возвращать значение. Если есть возвращаемое значение, то оно указывается в инструкции *return*.

Рассмотрим простую функцию:

```
def msum(x, y):
  return x + y

k = msum(3,5)
print(k)
```

Мы только что создали функцию *msum()* с двумя параметрами – ***x*** и ***y***. Результат работы этой функции – сумма переданных ей параметров ***x*** и ***y***. Далее мы вызвали функцию с параметрами (3, 5) и присвоили результат ее выполнения переменной **k**. Последний оператор выводит значение **k** (8 в нашем случае).

При вызове функции без параметров (если они определены при объявлении функции) вы получите сообщение:

```
Traceback (most recent call last):
  File "<pyshell#109>", line 1, in <module>
    msum()
TypeError: msum() missing 2 required positional arguments: 'x'
and 'y'
```

Вы можете создать ссылку на функцию, например:

```
s = msum
k = s(3, 5)
```

Обратите внимание, что при создании ссылки не нужно указывать круглые скобки после имени функции, иначе интерпретатор "подумает", что вы хотите вызвать функцию без параметров.

Можно также передать ссылку на функцию в качестве параметра другой функции. Функции, которые передаются по ссылке, называются функциями обратного вызова (*callback*). Пример:

```
def msum(x, y)
   return x + y

def fsum(f, x, y):
   return f(x, y)

k = fsum(msum, 3, 5)
```

11.2. Необязательные параметры функции

Ранее было показано, что, если вызвать функцию без параметров или с

меньшим количеством параметров, то будет выведено сообщение об ошибке. В Python вы можете задать параметры по умолчанию (необязательные параметры). Например:

```
def msum(x, y=1):
   return x + y

k = sum(3)      # результатом будет 4
```

Как видите, если второй параметр не задан, то его значение будет равно 2. Обратите внимание, что все необязательные параметры должны следовать после обязательных. Смешивать их нельзя, иначе получите сообщение об ошибке.

До этого мы использовали только позиционное присваивание параметров, например:

```
msum(4, 5)
```

В этом случае параметру ***x*** будет передано значение 4, а параметру ***y*** – 5. Но в Python мы можем использовать сопоставление по ключам, например:

```
>>> msum(y=3, x=2)
5
```

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ *.

```
>>> l1 = (2, 3)
>>> msum(*l1)
5
```

Количество элементов в словаре должно быть равно количеству параметров, которые может принимать функция.

Если значения параметров содержатся в словаре, то перед именем словаря нужно указать две звездочки**:

```
>>> d = {"x": 5, "y": 6}
>>> msum(**d)
11
```

Если вы указываете переменную в качестве значения параметра функции и передаваемый объект относится к неизменяемым типам, то, если функция изменяет значение параметра, это никак не отобразится на исходной переменной:

```
>>> def test(x):
   x = 1
   return x

>>> y = 2
>>> test(y)
1
>>> y
```

```
2
>>>
```

Однако функция может изменять значения объектов изменяемых типов, к которым относятся списки и словари.

11.3. Переменное число параметров

Представим, что вам нужно написать функцию, принимающую любое число параметров. Для этого используйте аргумент *. Пример:

```
def avg(first, *rest):
    return (first + sum(rest)) / (1 + len(rest))
```

Рассмотрим пример использования функции:

```
print(avg(1, 2))     # 1.5
print(avg(1, 2, 3, 4))    # 2.5
```

В данном случае **rest** – это кортеж, содержащий все дополнительно переданные аргументы. Наш код считает его последовательностью и работает как с последовательностью.

Чтобы принять любое число именных параметров (*keyword arguments*), используйте параметр, который начинается с **. Например:

```
import html

def make_element(name, value, **attrs):
  keyvals = [' %s="%s"' % item for item in attrs.items()]
  attr_str = ''.join(keyvals)
  element = '<{name}{attrs}>{value}</{name}>'.format(
            name=name,
            attrs=attr_str,
            value=html.escape(value))
  return element
```

Примеры использования:

```
# Создаем '<item size="large" quantity="6">Bus</item>'
make_element('item', 'Bus', size='large', quantity=6)

# Создаем '<p>&lt;Car&gt;</p>'
make_element('p', '<Car>')
```

Здесь **attrs** – словарь, который хранит переданные именные аргументы (если они были переданы, конечно).

Если вы хотите написать функцию, которая сможет принимать любое число позиционных и именных параметров, используйте * и ** вместе. Например:

```
def anyargs(*args, **kwargs):
  print(args)  # Кортеж
  print(kwargs)  # Словарь
```

С этой функцией все позиционные аргументы будут помещены в кортеж **args**, а все именные аргументы будут помещены в словарь **kwargs**.

Параметр * может быть указан исключительно как последний позиционный параметр в определении функции. Параметр ** тоже может появиться как последний параметр. Тонкий аспект определения функции – это то, что параметры могут все еще появиться после параметра *:

```
def a(x, *args, y):
pass
def b(x, *args, y, **kwargs):
pass
```

11.4. Анонимные функции

Некоторые функции, например, функции сортировки, подразумевают передачу в качестве параметров пользовательских функций, определяющих порядок сортировки или что-либо еще. В таких случаях удобнее использовать короткие встроенные функции, а не создавать полноценные функции оператором **def**.

Простые функции, которые делают ни что иное, как просто вычисляют выражение, могут быть заменены выражением *lambda*. Например:

```
>>> add = lambda x, y: x + y
>>> add(2,2)
4
>>> add('hello', 'world')
'helloworld'
>>>
```

Использование *lambda* здесь аналогично следующим кодом:

```
>>> def add(x, y):
...    return x + y
...
>>> add(2,2)
4
>>>
```

Как правило, *lambda* используется в контексте некоторой другой операции, такой как сортировка или сокращение данных:

```
>>> names = ['John', 'Den', 'Mark', 'Jane']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Den', 'Jane', 'John', 'Mark']
>>>
```

Хотя *lambda* позволяет вам определять простую функцию, определять такую функцию не рекомендуется. В частности, может быть определено только единственное выражение, результатом которого является возвращаемое значение. Это означает, что никакие другие функции языка, включая многократные операторы, условные выражения, итерация и обработка исключений не могут быть включены в эту функцию.

Вы можете написать много Python-кода, вообще не используя *lambda*. Однако вы будете иногда встречаться с ней в программах, где кто-то пишет много крошечных функций, которые вычисляют различные выражения или в программах, которые требуют, чтобы пользователи предоставили callback-функции.

Рассмотрим поведение следующего кода:

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

Теперь задайте себе вопрос. Каковы значения a(10) и b(10)? Если вы думаете, что 20 и 30, то вы ошибаетесь.

```
>>> a(10)
30
>>> b(10)
30
>>>
```

Проблема здесь в том, что значение *x* в выражении *lambda* является свободной переменной, которая связывается во время выполнения, а не во время определения. Поэтому значение *x* в *lambda*-выражении – то же, что и значение переменной *x* во время выполнения. Например:

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

Если вы хотите написать анонимную функцию, которая получает значение на момент определения и хранить ее, добавьте значение как значение по умолчанию, например:

```
>>> x = 10
>>> a = lambda y, x=x: x + y
>>> x = 20
>>> b = lambda y, x=x: x + y
>>> a(10)
20
```

```
>>> b(10)
30
>>>
```

Теперь поговорим о переносе дополнительного состояния в функциях обратного вызова. Представим, что вы написали код, который основывается на использовании callback-функций (например, обработчики событий), но вы хотите, чтобы callback-функция хранила дополнительную информацию о состоянии для использования внутри функции.

Этот пример связан с использованием функций обратного вызова, которые используются во многих библиотеках и структурах – особенно связанных с асинхронной обработкой. Для иллюстрации и из соображений тестирования определим следующую функцию, которая вызывает функцию обратного вызова:

```
def apply_async(func, args, *, callback):
  # Вычисляем результат
  result = func(*args)
  # Вызываем callback-функцию и передаем ей результат
  callback(result)
```

На практике такой код мог бы выполнять сортировку с использованием потоков, процессов и таймеров, но здесь мы не об этом. Вместо этого мы просто фокусируемся на вызове callback-функции. Вот пример, показывающий, как можно использовать предыдущий код:

```
>>> def print_result(result):
...   print('Result:', result)
...
>>> def add(x, y):
...   return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Result: helloworld
>>>
```

Как видите, функция *print_result()* принимает только один аргумент, который является результатом. Никакая другая информация не передается в нее. Такой недостаток информации иногда может представлять проблему,

например, когда вы хотите, чтобы функции обратного вызова взаимодействовала с другими переменными или частями среды.

Один из способов хранить дополнительную информацию в функции обратного вызова – использовать метод (и, соответственно, класс) вместо функции. Например, следующий класс хранит внутренний номер последовательности, который вызывается при каждом вызове метода *handler()*:

```
class ResultHandler:
  def __init__(self):
    self.sequence = 0
  def handler(self, result):
    self.sequence += 1
    print('[{}] Результат: {}'.format(self.sequence, result))
```

Чтобы использовать этот класс, вам нужно создать экземпляр и использовать связанный метод **handler** в качестве функции обратного вызова:

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Result: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Result: helloworld
>>>
```

11.5. Функции-генераторы

Функция-генератор – это функция, которая может возвращать одно значение из нескольких значений на каждой итерации. Превратить функцию в генератор позволяет ключевое слово *yield*. Рассмотрим пример простой функции-генератора:

```
def gen(x, y):
  for i in range(1, x+1):
    yield i + y
```

Вот как можно использовать генератор:

```
>>> s = gen(3,3)
>>> print(s.__next__())
4
>>> print(s.__next__())
5
>>> print(s.__next__())
6
>>> print(s.__next__())
Traceback (most recent call last):
  File "<pyshell#135>", line 1, in <module>
    print(s.__next__())
StopIteration
>>>
```

Как видите, при каждом следующем запуске функция увеличивает результат предыдущей операции на 1. Максимальное число итераций устанавливается вторым параметром, а первый параметр – начальное число, которое будет постепенно увеличиваться на 1. Четвертый вызов функции завершился с ошибкой *StopIteration*, поскольку второй параметр задает максимальное число итераций, равное трем.

11.6. Декораторы

Основное назначение декораторов – выполнить какие-либо действия перед выполнением функции. Рассмотрим пример:

```
def deco(f):
   print("my_func is running")
   return f
@deco
def my_func(x):
   return x * 2

print(my_func(5))
```

11.7. Рекурсия

Рекурсия – это явление, когда функция вызывает саму себя. В современном программировании рекомендуется не использовать рекурсию и пытаться любой рекурсивный алгоритм заменить нерекурсивным. Опасность рекурсии в том, что вы можете забыть предусмотреть условие выхода из рекур-

сии, и тогда функция будет запускать себя снова и снова и ничего хорошего из этого не получится.

Классическим примеров рекурсивной функции является функция вычисления факториала:

```
def fact(n):
   if n == 0 or n == 1: return 1
   else:
      return n * fact(n - 1)
```

Как видите, мы предусмотрели условие выхода из рекурсии – если n = 0 или n = 1, то функция просто возвращает 1. В противном случае функция вызывает саму себя, передав значение **n**, уменьшенное на 1. Следовательно, при каждом вызове функции значение параметра будет уменьшаться, а когда оно станет равно 1, функция просто вернет 1.

Ради справедливости нужно отметить, что функция вычисления факториала называется *factorial()* и имеется в модуле **math**.

11.8. Глобальные и локальные переменные

Глобальные переменные – это все переменные, объявленные за пределами функции. Локальные переменные – это переменные, объявленные в самой функции.

11.8.1. ИНКАПСУЛЯЦИЯ

А зачем функции возвращаю значения? Посмотрим на такую функцию:

```
def fun():
  res = 10
  return res
```

Почему бы нам не обратиться к переменной **res** напрямую – в коде нашей программы? Спешу вас огорчить: потому что нельзя. Переменная **res** не

существует вне функции. Вообще ни одна переменная, созданная внутри функции (в том числе и параметры), извне не доступна. Данная техника называется инкапсуляцией. Она помогает сохранить независимость отдельных фрагментов кода. Параметры и возвращаемые значения используются, чтобы передавать важную информацию и игнорировать все прочее. За значениями переменных, созданных внутри функции, не нужно следить во всем остальном коде, что очень удобно. И чем больше программа, тем более заметно это преимущество.

Поскольку код функции скрыт от основной программы и от других функций, в разных функциях вы можете использовать одни и те же имена переменных, например, res для обозначения результата.

11.8.2. ОБЛАСТЬ ВИДИМОСТИ. КЛЮЧЕВОЕ СЛОВО *GLOBAL*

Благодаря инкапсуляции, функции как бы закрыты от основной программы и от других функций. Пока мы знаем только единственный механизм обмена информацией между ними – это параметры и возвращаемые значения. Однако есть еще и другой способ – глобальные переменные.

Область видимости – это способ представления разных частей программы, отделенных друг от друга.

Рассмотрим небольшой пример:

```
def fun1():
    res = 10
    print(res)

def fun2()
    res = 20
    print(res)

res = 30
print(res)
fun1()
fun2()
print(res)
```

Вывод программы будет таким:

```
30
10
20
30
```

Сначала мы определили две функции, внутри каждой из них переменной **res** присваивается разное значение – 10 и 20 соответственно. Далее в основной программе мы определили переменную **res** со значением 30.

Сначала мы выводим значение переменной **res** до запуска функций. Затем запускаем обе функции и снова выводим значение переменной **res**, чтобы убедиться, что ни одна из функций его не изменила.

Как видите, программа и две наших функции выводят собственные значения переменной **res**, а все потому что у нас есть целых три области видимости – одна глобальная (программа) и две локальные – по одной для каждой из функций.

Любая переменная, созданная в глобальной области видимости, называется *глобальной*. Переменная, объявленная в локальной области, называется *локальной*.

Если вам нужно получить доступ к глобальной переменной, тогда нужно использовать ключевое слово *global*. Изменим нашу программу так:

```
def fun1():
    global res
    print(res)

def fun2():
    res = 20
    print(res)

res = 30
print(res)
fun1()
fun2()
print(res)
```

Теперь вывод программы будет таким:

```
30
30
20
```

```
30
```

Посмотрим, что произошло. Сначала мы вывели значение **res**, определенное в основной программе. Затем мы вызвали функцию *fun1()*, которая благодаря ключевому слову *global* получила доступ к глобальной переменной **res** и вывела ее значение. Функция *fun2()* вывела собственное значение **res**. Далее мы отобразили значение переменной **res** из глобальной области.

Использование ключевого слова *global* позволяет не только читать, но и записывать, то есть изменять значение глобальной переменной. Рассмотрим следующий пример:

```
def fun1():
    global res
    res = 50
    print(res)

def fun2():
    global res
    print(res)

res = 30
print(res)
fun1()
fun2()
print(res)
```

Изначально значение глобальной переменной **res** было 30. Затем в функции *fun1()* мы изменили его на 50. Функция *fun2()*, поскольку она вызывается после функции *fun1()*, получает уже новое значение – 50. Поскольку функция *fun1()* изменила значение переменной res() в глобальной области, то последний оператор *print()* отобразит также 50. В итоге вывод программы будет таким:

```
30
50
50
50
```

11.8.3. СТОИТ ЛИ ИСПОЛЬЗОВАТЬ ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ?

Да, в Python вы можете использовать глобальные переменные. А нужно ли? Ведь, по сути, тогда вы лишаетесь преимуществ инкапсуляции – вам нужно будет следить за значением переменной. Одна логическая ошибка в боль-

шой программе приведет к непоправимым последствиям и многочасовой отладке. Нужно ли вам это? Глобальные переменные только запутывают код, поскольку за их постоянно меняющимися значениями сложно следить. Поэтому постарайтесь ограничить их использование по максимуму. Основной девиз должен быть таким: если можно обойтись без глобальной переменной, сделайте это.

11.9. Документирование функций

Очень полезно документировать функции по мере их написания. Пройдет время и вы даже не сможете вспомнить, как работает та или иная функция, какие параметры она должна принимать. Конечно, в простом случае достаточно взглянуть на ее код и все станет понятно, но есть ситуации, когда функции содержат сотни строк кода. В таких ситуациях на помощь приходит документирование.

В Python есть особый механизм, который называется документирующими строками. Такие строки представляют собой строку в тройных кавычках. В блоке кода документирующая строка должна обязательно идти первой по порядку. Рассмотрим пример:

```
def warning(message):
    """ Выводит сообщение, заданное параметром message,
    обрамленное символами * для привлечения внимания """
    print("*" * 10, message, "*" * 10)
```

Данная строка воспринимается как многострочный комментарий и никак не обрабатывается интерпретатором, тем более не выводится на экран и не возвращается в качестве значения функции.

При желании документировать функцию можно и с помощью комментариев. Но использование документирующих строк элегантнее и удобнее.

11.10. Возвращаем несколько значений.

Иногда нужно, чтобы функция вернула несколько значений. Для этого просто возвращайте кортеж, например:

```
def fun():
```

```
    return 1, 2, 3

a, b, c = fun()

print(a, b, c)
```

В результате будет выведено

```
1 2 3
```

Хотя кажется, что функция *fun()* возвращает несколько значений, на самом деле она возвращает одно значение, но в виде кортежа. Это выглядит немного странным, но кортеж формирует запятая, а не круглые скобки. Пример:

```
>>> a = (1, 2)    # Со скобками
>>> a
(1, 2)
>>> b = 1, 2   # Без скобок
>>> b
(1, 2)
>>>
```

При вызове функций, которые возвращают кортеж, принято присваивать результат несколько переменным, как было показано выше. Это просто – распаковка кортежа. Возвращаемое значение можно также присвоить одной переменной.

```
>>> x = fun()
>>> x
(1, 2, 3)
>>>
```

11.11. Именованные аргументы

Давайте рассмотрим функцию *hello*, использующую самый простой способ передачи параметров:

```
def hello(name, city):
  print("Привет, ", name, "! Мы едем в ", city)
```

Ничего сложного. Мы просто определили два параметра – **name** и **city**. Первый – имя, второй город. Такие параметры называются позиционными, поскольку строго определен порядок их следования. Значения функция принимает в том же порядке, что и указаные позиционные параметры.

Рассмотрим вызов функции:

```
hello("Марк", "Санкт-Петербург")
```

Вывод будет таким:

```
Привет, Марк! Мы едем в Санкт-Петербург
```

А что будет, если программист перепутает порядок следования аргументов:

```
hello("Санкт-Петербург", "Марк")
```

Тогда параметру **name** будет передано значение "Санкт-Петербург", а параметру **city** – "Марк". В итоге вывод будет не таким, как мы ожидали:

```
Привет, Санкт-Петербург! Мы едем в Марк
```

Это у нас еще простой случай, а представьте, если бы второй параметр был числом и далее шла его обработка как числа. Тогда мы бы получили ошибку и выполнение программы было бы остановлено!

Специально для таких случаев предназначены ***именованные аргументы***. Они позволяют указывать аргументы в любом порядке при условии, что мы задаем имя аргумента. Вызовем функцию так:

```
hello(city = "Санкт-Петербург", name = "NoName")
```

Преимущества использования именованных аргументов следующие:

- Ясность – вы знаете, какое значение и какому параметру передаете. Даже если вы будете указывать параметры в том же порядке, в котором они и объявлены, использование именованных аргументов добавляет прозрачности в вашу программу.
- Возможность изменения порядка следования параметров – если вы намерено изменили порядок следования аргументов (потому что вам так захотелось) или случайно перепутали его, ничего страшного не произой-

дет, и функция будет работать корректно (в отличие от использования позиционных параметров).

11.12. Практический пример: программа для чтения RSS-ленты

Формат новостной ленты RSS довольно популярен во всем мире, особенно на новостных сайтах и всевозможных форумах. Многие пользователи предпочитают читать RSS-ленту, а не заходить на сайт. Почему? Да потому что в RSS-ленту не попадает реклама и прочий не нужный пользователю контент – он видит только текст новости и некоторые другие служебные данные (дату и время публикации, имя автора и т.д.)

Сейчас мы попробуем написать программу, которая будет читать RSS новостную ленту (лист. 11.1).

Листинг 11.1. Программа для чтения RSS

```
def rss_reader(url):
    from urllib.request import urlopen
    from xml.etree.ElementTree import parse

    # Загружаем RSS-ленту и парсим ее
    u = urlopen(url)
    doc = parse(u)
    # Извлекаем и выводим итересующие теги
    for item in doc.iterfind('channel/item'):
        title = item.findtext('title')
        date = item.findtext('pubDate')
        link = item.findtext('link')

        print(title)
        print(date)
        print(link)
        print()

rss_reader('http://example.com/rss.php')
```

Итак, у нас есть функция *rss_reader()*, которой нужно передать адрес новостной ленты. Посмотрим, что происходит внутри функции. Первым делом импортируются модули **urlopen** и **parse**. Первый нужен для открытия

удаленного документа, второй – для разбора (парсинга XML-формата, в котором и распространяется новостная лента).

Далее в переменной **u** мы читаем новостную ленту, адрес которой задается параметром *url*. После мы выполняем парсинг этой ленты и результат сохраняем в *doc*.

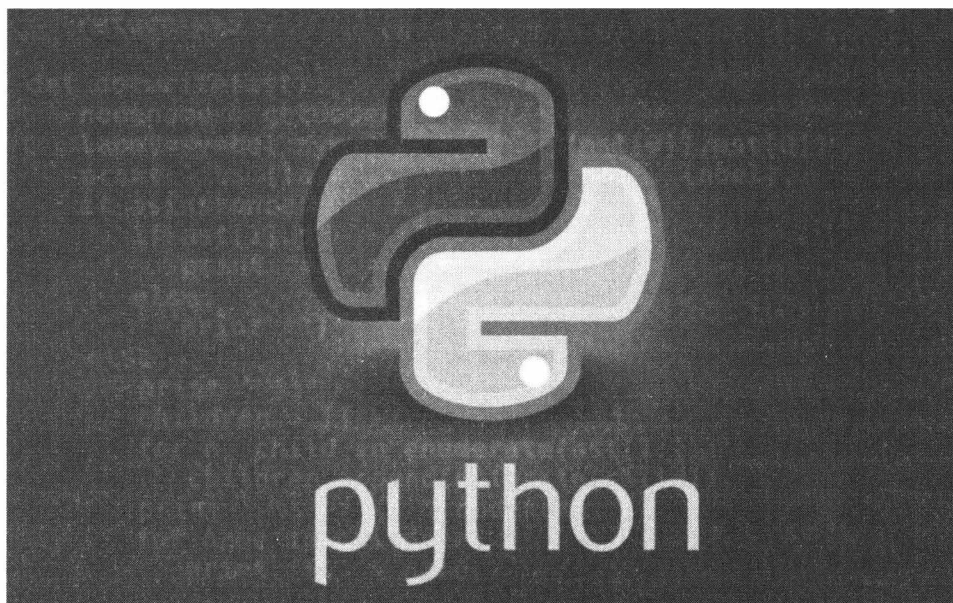
Переменная doc содержит уже "разобранную" новостную ленту. Функция xml.etree.ElementTree.parse() парсит весь XML-документ в объект документа. Вы можете использовать методы вроде *find()*, *iterfind()* и *findtext()* для поиска определенных XML-документов. Аргументы к этим функциям – имена определенных тегов, вроде channel/item или title.

При определении тегов вы должны принять полную структуру документа во внимание. Каждая операция **find** работает относительно начального элемента. Аналогично, имя тега, которое вы передаете каждой операции, тоже указывается относительно начального элемента. В примере вызов к doc.iterfind('channel/item') находит все элементы "item", которые находятся внутри элемента "channel". "doc" представляет верхнюю часть документа (элемент "rss"). Более поздние вызовы *item.findtext()* будут иметь место относительно найденных элементов "item".

Далее в цикле мы просто находим и выводим элементы title, pubDate, link. Как правило, элементы с такими названиями есть в большинстве случаев, но вы можете просмотреть код RSS-ленты и изменить названия элементов, если в коде используются другие.

ГЛАВА 12.

ДАТА И ВРЕМЯ



12.1. Получение текущей даты и времени

В Python для работы с датой и временем используются модули **time**, **datetime**, **calendar**, **timeit**. Первый модуль позволяет получить текущую дату и время, а также произвести форматированный вывод времени/даты. Второй модуль используется для манипулирования датой и временем. Третий модуль позволяет вывести календарь в виде простого текста или HTML-кода. Последний модуль позволяет измерять время выполнения фрагментов кода – он используется для оптимизации программы.

Начнем с модуля **time**. Функция **time** позволяет получить число секунд, прошедшее с 1 января 1970 года (эта дата считается началом эпохи UNIX-систем, поэтому во многих языках программирования считается начальной):

```
>>> import time
>>> time.time()
1528374820.9274228
```

Понятно, что такая дата будет удобна лишь самому компьютеру, но не человеку. Пока вы высчитаете, какому времени соответствует выданный функцией результат, время существенно изменится. Поэтому для работы с датой и временем лучше использовать другую функцию, например, *gmtime()*, которая возвращает объект *struct_time*, представляющий универсальное время UTC. Функции передается единственный параметр – количество секунд, прошедшее с 1 января 1970 года, то есть то, что возвращает функция *time()*:

```
>>> t = time.time()
>>> time.gmtime(t)
time.struct_time(tm_year=2018, tm_mon=6, tm_mday=7, tm_
hour=12, tm_min=33, tm_sec=54, tm_wday=3, tm_yday=158, tm_
isdst=0)
```

Как видите, это структура. Использовать структуру нужно так:

```
>>> tm = time.gmtime(t)
>>> tm.tm_hour
12
```

Если нужно не UTC-время, а местное время, тогда нужно использовать функцию *localtime()*, которая также принимает также количество секунд, прошедшее с 1 января 1970 года. Если нужно работать с текущим временем, то ни в *localtime()*, ни в gmtime() не нужно вообще ничего передавать – по умолчанию будет использовано локальное время. Функция *localtime()*, как и *gmtime()*, возвращает объект *struct_time*, содержащий следующие атрибуты:

- tm year - 0 – год;
- tm_mon - 1 – месяц (число от 1 до 12);
- tm_mday - 2 – день месяца (число от 1 до 31);
- tm_hour - 3 – час (число от 0 до 23);
- tm_min - 4 – минуты (число от 0 до 59);
- tm_sec - 5 – секунды (число от 0 до 59);
- tm wday - 6 – день недели (число от 0 (для понедельника) до 6 (для воскресенья));

- tm yday - 7 – количество дней, прошедшее с начала года (число от 1 до 366);
- tm_isdst - 8 – флаг коррекции летнего времени (значения 0, 1 или -1).

12.2. Форматирование даты и времени

Понятно, что можно проанализировать объект *scruct_time* и отобразить информацию о времени/дате так, как вам нужно. Но гораздо проще не заниматься этим вручную, а поручить форматирование даты и времени специальным функциям. Например, функция *strftime()* возвращает строковое представление даты в соответствии с заданной строкой формата:

```
strftime(<строка формата>[, <объект scruct_time>])
```

Первый параметр – это строка формата, мы ее рассмотрим чуть позже. Второй параметр – необязательный. Если он не указан, то будет использована текущая дата (время). Как именно будет отображаться дата и время, зависит от локали. Рассмотрим несколько примеров:

```
>>> time.strftime("%d.%m.%Y")
'22.03.2021'
>>> time.strftime("%H:%M:%S")
'15:34:56'
```

Для обратного преобразования, то есть строки, содержащей дату в объект struct_time, используется функция *strptime()*:

```
strptime(<строка с датой>[, <строка формата>])
```

Формат можно не указывать (второй параметр). Первый параметр – это строка с датой/временем, которая будет преобразована в объект *struct_time*, который и будет возвращен функцией. Если строка не соответствует формату, вы получите исключение *ValueError*.

В строке формата могут использоваться следующие модификаторы:

- **%a** – аббревиатура дня недели в зависимости от настроек локали;
- **%A** – название дня недели в зависимости от настроек локали;
- **%m** – номер месяца с предваряющим нулем (от "01" до "12");
- **%b** – аббревиатура месяца в зависимости от настроек локали (например, "feb" для января);
- **%B** – название месяца в зависимости от настроек локали (например, "March");
- **%d** – номер дня в месяце с предваряющим нулем (от "01" до "31 ");
- **%j** – день с начала года (от "001" до "366");
- **%U** – номер недели в году (от "00" до "53"). Неделя начинается с воскресенья. Все дни с начала года до первого воскресенья относятся к неделе с номером 0;
- **%W** – номер недели в году (от "00" до "53"). Неделя начинается с понедельника. Все дни с начала года до первого понедельника относятся к неделе с номером 0;
- **%w** – номер дня недели ("0"– для воскресенья, "6"–для субботы);
- **%H** – часы в 24-часовом формате (от "00" до "23");
- **%I** – часы в 12-часовом формате (от "01" до "12");
- **%M** – минуты (от "00" до "59");
- **%S** – секунды (от "00" до "59");
- **%p** – эквивалент значениям AM и PM в текущей локали;
- **%c** – представление даты и времени в текущей локали;
- **%x** – представление даты в текущей локали;
- **%X** – представление времени в текущей локали;
- **%y** – год из двух цифр (от "00" до "99");
- **%Y** – год из четырех цифр (например, "2021");
- **%Z** – название часового пояса или пустая строка;
- **%%** – символ "%".

Прежде, чем форматировать дату и время, нужно импортировать локаль:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, ('russian'))
'Russian_Russia.1251'
```

После этого можно работать с датой и временем:

```
>>> print(time.strftime("%A %d %b %Y %H:%M:%S\n%d.%m.%Y"))
пятница 19 мар 2021 18:42:47
19.03.2021
```

12.3. Модуль calendar

Модуль **calendar** можно использовать для вывода календаря в текстовом формате или в HTML. В этом модуле вы найдете следующие классы:

- **Calendar** – базовый класс, который наследуют все остальные классы
- **TextCalendar** – текстовый календарь
- **HTMLCalendar** – календарь в формате HTML
- **LocaleTextCalendar** – позволяет вывести календарь на языке указанной локали
- **LocaleHTMLCalendar** – то же самое, что и LocaleHTMLCalendar, но выводит календарь в формате HTML

Каждому из этих классов нужно передать первый день недели. "Локализированным" классам нужно также еще передать название локали.

```
>>> import calendar
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2021))
```

Данный код выводит календарь на 2021 год на русском языке в текстовом формате. Результат работы программы приведен на рис. 12.1.

Аналогично, вы можете использовать класс LocaleHTMLCalendar, чтобы создать календарь в формате HTML.

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help

        Апрель                     Май                      Июнь
Пн Вт Ср Чт Пт Сб Вс      Пн Вт Ср Чт Пт Сб Вс      Пн Вт Ср Чт Пт Сб Вс
       1  2  3  4                      1  2          1  2  3  4  5  6
 5  6  7  8  9 10 11       3  4  5  6  7  8  9       7  8  9 10 11 12 13
12 13 14 15 16 17 18      10 11 12 13 14 15 16      14 15 16 17 18 19 20
19 20 21 22 23 24 25      17 18 19 20 21 22 23      21 22 23 24 25 26 27
26 27 28 29 30            24 25 26 27 28 29 30      28 29 30
                          31

        Июль                     Август                   Сентябрь
Пн Вт Ср Чт Пт Сб Вс      Пн Вт Ср Чт Пт Сб Вс      Пн Вт Ср Чт Пт Сб Вс
          1  2  3  4                         1             1  2  3  4  5
 5  6  7  8  9 10 11       2  3  4  5  6  7  8       6  7  8  9 10 11 12
12 13 14 15 16 17 18       9 10 11 12 13 14 15      13 14 15 16 17 18 19
19 20 21 22 23 24 25      16 17 18 19 20 21 22      20 21 22 23 24 25 26
26 27 28 29 30 31         23 24 25 26 27 28 29      27 28 29 30
                          30 31

       Октябрь                   Ноябрь                   Декабрь
Пн Вт Ср Чт Пт Сб Вс      Пн Вт Ср Чт Пт Сб Вс      Пн Вт Ср Чт Пт Сб Вс
             1  2  3       1  2  3  4  5  6  7             1  2  3  4  5
 4  5  6  7  8  9 10       8  9 10 11 12 13 14       6  7  8  9 10 11 12
11 12 13 14 15 16 17      15 16 17 18 19 20 21      13 14 15 16 17 18 19
18 19 20 21 22 23 24      22 23 24 25 26 27 28      20 21 22 23 24 25 26
25 26 27 28 29 30 31      29 30                     27 28 29 30 31

>>>
                                                               Ln: 59 Col: 4
```

Рис. 12.1. Календарь в текстовом виде

12.4. Функция sleep

В модуле **time** есть очень полезная функция – *sleep()*, позволяющая приостановить выполнение сценария на указанное в секундах время. Например:

```
import time as t
t.sleep(10)
```

В данном случае выполнение сценария будет приостановлено на 10 секунд.

12.5. Измерение времени выполнения фрагментов кода

Модуль timeit содержит очень полезные функции, которые можно использовать для оптимизации работы программы, а именно для определения, сколько времени выполняется тот или иной фрагмент кода.

Используя модуль **timeit**, вы можете определить "узкие" места в производительности вашей программы.

Измерение производительности производятся с помощью класса Timer. Конструктор класса Timer выглядит так:

```
Timer([stmt='код'][, setup='код_настройки'][, timer=<функция_
таймера>])
```

Первый параметр – это код, время работы которого нужно измерить. Второй параметр – это код, который должен быть выполнен до измеряемого кода. Здесь, например, можно загрузить необходимые для выполнения указанного в первом параметре кода модули. Третий параметр задает функцию таймера.

Метод **timeit** объекта Timer позволяет измерить время выполнение кода. Параметр **number** задает количество выполнений кода, например:

```
timeit(number=1000)
```

По умолчанию код **setup** выполняется один раз, а код **stmt** – один миллион раз, если иного не задано параметром **number**.

Метод *repeat()* позволяет вызвать метод *timeit()* несколько раз. Его параметры по умолчанию выглядят так:

```
repeat(repeat=3, number=1000000)
```

Первый параметр задает, сколько раз будет вызван *timeit()*, второй – значение **number**, которое будет передано в **timeit**. Метод возвращает результат для каждого выполнения *timeit()*.

Типичное использование всего этого выглядит так:

```
t = Timer(...)        # за пределами try/except
try:
    t.timeit(...)     # или t.repeat(...)
except:
    t.print_exc()
```

Пример:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample
string"; char = "g"')
>>> t.timeit()
0.14950477718537059
>>> t.repeat()
[0.11337469360387331, 0.11169325663377094,
0.11340548288711716]
>>>
```

Зачем все усложнять? Ведь теоретически, можно написать такой код:

```
from time import *
t1 = time()
# Код, время выполнения которого нужно измерить
t2 = time()
t = t2 - t1
```

В результате в **t** будет содержаться время выполнения кода в секундах. Но такой способ не даст таких точных измерений, как метод **timeit**, хотя бы потому что он считает, сколько времени прошло в общем, а ни сколько заняло выполнение кода. Рассмотрим пример:

```
>>> from time import *
>>> t1 = time()
>>> print('Привет')
Привет
>>> t2 = time()
>>> t2 - t1
27.691545009613037
>>>
```

Неужели на вывод строки *Привет* было потрачено 27 секунд? Конечно же нет. В среде IDLE этот метод вообще использовать нельзя, поскольку он учитывает не только время выполнения кода, но и время набора этого кода программистом. В реальных программах данный метод сгодится разве что для грубого оценивания больших фрагментов кода, где сотые секунды – не важны.

12.6. Модуль datetime

Для выполнения преобразований и вычислений, вовлекающих единицы времени, в Python используется модуль **datetime**. Например, чтобы представить интервал времени, создайте экземпляр *timedelta*:

```
>>> from datetime import timedelta
>>> a = timedelta(days=3, hours=4)
>>> b = timedelta(hours=6.5)
>>> c = a + b
>>> c.days
3
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
```

Обратите внимание: свойства у *hours* нет, вместо него нужно разделить свойство *seconds* на 3600.

Если вам нужно представить специфические даты и время, создайте экземпляр **datetime** и используйте стандартные математические операции для манипуляции с ним. Например:

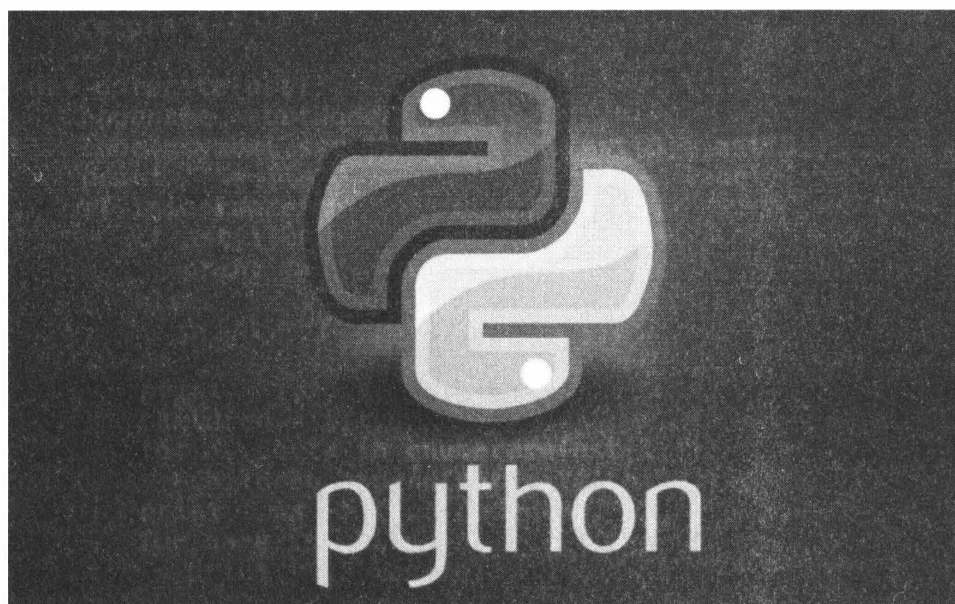
```
>>> from datetime import datetime
>>> from datetime import timedelta
>>> a = datetime(2021, 3, 20)
>>> print(a + timedelta(days=5))
2021-03-25 00:00:00
>>> b = datetime(2021, 7, 27)
>>> d = b - a
>>> d.days
129
>>> now = datetime.today()
>>> print(now)
2021-03-19 19:00:26.413132
>>> print(now + timedelta(days=2))
2021-03-21 19:00:26.413132
>>>
```

```
>>> from datetime import datetime
>>> from datetime import timedelta
>>> a = datetime(2021, 3, 20)
>>> print(a + timedelta(days=5))
2021-03-25 00:00:00
>>> b = datetime(2021, 7, 27)
>>> d = b - a
>>> d.days
129
>>> now = datetime.today()
>>> print(now)
2021-03-19 19:00:26.413132
>>> print(now + timedelta(days=2))
2021-03-21 19:00:26.413132
>>>
```

Рис. 12.2. Практическое использование модуля datetime

ГЛАВА 13.

МОДУЛИ И ПАКЕТЫ



13.1. Понятие модуля

Модулем в Python называется любой файл с программой. Каждый модуль может импортировать в себя другой модуль, в результате чего он получает доступ к идентификаторам, находящимся в другом модуле.

Получить имя модуля можно с помощью специального атрибута __name__. Ранее у нашей программы был только один модуль (к которому мы, возможно, подключали другие модули) – __main__. Проверить, что мы находимся в __main__ можно так:

```
if __name__ == "__main__":
  <сделать что_то>
```

13.2. Инструкция import

Для подключения другого модуля используется инструкция *import*, в которой мы уже все знакомы:

```
import <название модуля>
```

Например:

```
import itertools
```

Затем обратиться к идентификатору, находящемуся в модуле можно так:

```
itertools.count()
```

Однако имена некоторых модулей слишком длинные, и чтобы сделать код компактнее, вы можете использовать псевдонимы модулей. Псевдоним задается с помощью конструкции:

```
import <модуль> as <псевдоним>
```

Например:

```
import itertools as it
```

После этого все идентификаторы модуля *itertools* будут доступны через псевдоним **it**:

```
it.count()
```

Вот еще пример:

```
import math as m
a = m.pi * 10
```

Мы подключили модуль **math**, создали псевдоним **m** и доступ к числу Пи теперь осуществляется через идентификатор m.pi.

Теперь немного практики. Создайте два файла – main.py и module.py. Во второй файл поместите указанный в листинге 13.1 код.

Листинг 13.1. Файл module.py

```
# -*- coding: utf-8 -*-
a = 0
```

В модуле module.py мы определили только один идентификатор – ***a***. Теперь создайте файл main.py (лист. 13.2).

Листинг 13.2. Файл main.py

```
# -*- coding: utf-8 -*-
import module as m

a = 2
print(a)
print(m.a)
input()
```

Сначала вы увидите число 2 – это значение идентификатора ***a*** в основной программе. А затем вы увидите число 0 – это значение идентификатора с таким же именем в модуле. Думаю, принцип понятен.

> ***Примечание***. Обратите внимание на содержимое папки с файлами после подключения модуля module.py. Внутри папки автоматически был создан каталог _pycache_ с файлом module.cpython-32.pyc. Этот файл содержит скомпилированный байт-код одноименного модуля. Байт-код создается при первом импортировании модуля и изменяется только после изменения кода внутри модуля.

13.3. Инструкция from

Инструкцию *from* удобно использовать для импорта только определенных идентификаторов. Синтаксис следующий:

```
from <название модуля> import <идентификатор> [as <псевдоним>]
```

Пример:

```
from math import pi
```

После этого к идентификатору **pi** можно ссылаться напрямую, без указания имени модуля и псевдонима:

```
a = pi * 10
```

Для импорта всех идентификаторов из модуля можно использовать звездочку *:

```
from <название модуля> import *
```

Посмотрим, что произойдет с нашей программой, если использовать инструкцию *from*. Измените main.py, чтобы он выглядел, как показано в листинге 13.3.

Листинг 13.3. Файл main.py

```
# -*- coding: utf-8 -*-
from module import *
a = 2
print(a)
input()
```

Из модуля импортируется идентификатор ***a*** и в программе есть идентификатор ***a***. Какое будет выведено значение? Здесь происходит слияние пространств и все идентификатор попадают в одно общее пространство. Если программа использует идентификатор с таким же именем, то она переопределит его значение, поэтому будет выведено значение 2.

При желании можно импортировать несколько идентификаторов. Как правило, это делается, если нужно импортировать только интересующие идентификаторы. Пример:

```
from math import (pi, floor, sin, cos)
```

13.4. Путь поиска модулей

В каких каталогах Python будет искать модули? В самом простом случае модули размещаются в одном каталоге с вашей программой. В этом случае нет необходимости настраивать пути поиска, поскольку текущий каталог автоматически включается в путь поиска.

Просмотреть текущий путь поиска можно с помощью следующих команд:

```
>>> import sys
>>> sys.path
['', 'E:\\Python\\Lib\\idlelib', 'E:\\Python\\python392.zip', 'E:\\
Python\\DLLs', 'E:\\Python\\lib', 'E:\\Python', 'E:\\Python\\lib\\
site-packages']
>>>
```

Путь поиска программ состоит из текущего каталога, пути поиска стандартных модулей, переменной окружения PYTHONPATH и содержимого pth-файлов (они должны находиться в каталоге Lib\site-packages, имя файла может быть любым, но расширение должно быть .pth). Один из самых простых способов добавить нужные каталоги в путь поиска – это создать pth-файл. Перейдите в каталог Lib\site-packages, создайте файл, скажем, paths.pth и добавьте в него нужные пути поиска – по одному в каждой строке:

```
C:\projects\my_libs
C:\python\my_py
```

Второй часто используемый способ пути поиска модулей – изменение переменной окружения PYTHONPATH.

13.5. Повторная загрузка модулей

Модуль загружается только один раз – при первой операции импорта. Все последующие вызовы *import* будут возвращать уже загруженный объект модуля, даже если сам модуль был изменен. Допустим, вы изменили модуль, как его загрузить заново?

Для этого нужно использовать функцию *reload()* из модуля **imp**:

```
from imp import reload
reload(<модуль>)
```

13.6. EGG-файлы

Сложные расширения (вроде Pytz) состоят из множества модулей, поэтому для простоты распространения таких пакеты принято распространять в виде EGG-файлов. Такие пакеты можно скачать на сайтах разработчиков пакетов-расширений. Разберемся, как их можно установить:

1. Скачайте egg-файл и поместите его в каталог c:\Python\Lib\site-packages\

2. Выполните из этого каталога команду:

```
python easy_install.py -Z <EGG-файл>
```

Например,

```
python easy_install.py -Z pytz.egg
```

Команда должна выглядеть именно так:

```
python easy_install.py -Z <EGG-файл>
```

А не так (как показано в некоторых руководствах):

```
python easy_install -Z <EGG-файл>
```

Ввод этой команды приведет к ошибке.

3. Просмотрите вывод команды. В случае успеха вы должны увидеть строку `Finished processing dependencies for <название модуля>`

13.7. Разделение модуля на несколько файлов

Пусть у нас есть модуль, который нам нужно разделить на несколько файлов. Необходимо это сделать, не повредив существующий код, сохраняя отдельные файлы объединенными как единственный логический модуль.

Программный модуль можно разделить на отдельные файлы, превратить его в пакет. Рассмотрим следующий простой модуль:

```
# mymodule.py
class A:
def test(self):
  print('A.test')
class B(A):
  def bar(self):
    print('B.bar')
```

Предположим, что вы хотите разбить mymodule.py на два файла – в каждом будет собственное определение класса. Чтобы сделать это, начните с замены файла mymodule.py каталогом *mymodule*. В этот каталог поместите два файла:

```
mymodule/
  __init__.py
  a.py
  b.py
```

В файл *a.py* поместите этот код:

```
# a.py
class A:
  def test(self):
    print('A.test')
```

В файл *b.py* поместите этот код:

```
# b.py
from .a import A
  class B(A):
    def bar(self):
      print('B.bar')
```

Наконец, в файл __init__.py поместите код, соединяющий все это вместе:

```
# __init__.py
from .a import A
from .b import B
```

Если вы сделаете эти действия, в результате будет пакет *mymodule*, который будет представлен как единственный логический модуль:

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.test()
A.test
>>> b = mymodule.B()
>>> b.bar()
B.bar
>>>
```

Нужно определиться, хотите ли вы, чтобы пользователи работали с большим количеством маленьких модулей или с одним большим модулем? Например, в большой базе кода, можно просто разбить большой модуль на несколько меньших, но в результате пользователи должны будут использовать много операторов импорта, например:

```
from mymodule.a import A
from mymodule.b import B
...
```

Это работает, но создает определенные неудобства для пользователя, который должен знать, где расположены различные части. Часто проще объединить все в один модуль и использовать единственный оператор импорта:

```
from mymodule import A, B
```

Для нашего последнего примера нужно думать о *mymodule* как об одном большом исходном файле. Однако этот раздел показывает, как объединить несколько файлов в единственное логическое пространство имен. Ключ к достижению этого результата – создание каталога пакета и использование файла __init__.py для объединения частей.

Когда модуль будет разделен, вы должны обратить особое внимание на перекрестные ссылки. Например, в этом разделе, *класс B* должен получить доступ к *классу A* как к базовому. Чтобы получить такой доступ, используется относительный импорт .a import A.

Всюду по разделу используется относительный импорт, чтобы не указывать жестко имя модуля верхнего уровня в исходном коде. В результате будет проще переименовать модуль или переместить его в другое место.

13.8. Создание отдельных каталогов импорта кода под общим пространством имен

На этот раз у нас есть много кода, разделенного на части и, возможно, обслуживаемого и распределяемого разными людьми. Каждая часть кода организована как каталог файлов, подобно пакету. Однако вместо того, чтобы

установить каждую часть как отдельный именованный пакет, вы хотите, чтобы все части были объединены под одним общим префиксом пакета.

По существу, задача заключается в том, что нужно определить пакет Python верхнего уровня, который служит пространством имен для большого количества отдельно сохраняемых подпакетов. Эта проблема часто возникает в больших фреймворках, где разработчики хотят поощрить пользователей распределять плагины или дополнительные пакеты.

Чтобы объединить отдельные каталоги под общим пространством имен, нужно организовать код в виде обычного пакета, но опустить файлы __init__.py в каталогах, где будут объединяться компоненты. Рассмотрим небольшой пример. Предположим, что у вас есть два разных каталога кода:

```
foo-package/
  test/
    id.py

bar-package/
  test/
    run.py
```

В этих каталогах имя *test* используется как общее пространство имен. Обратите внимание, что здесь нет файлов __init__.py – ни в одном из каталогов.

Теперь посмотрим, что произойдет, если вы добавите оба каталога – foo-package и bar-package – в модуль Python и попытаетесь вызвать некоторые операторы импорта:

```
>>> import sys
>>> sys.path.extend(['foo-package', 'bar-package'])
>>> import test.id
>>> import test.run
>>>
```

Вы заметите, что, словно по волшебству, два разных каталога объединятся вместе, и вы сможете импортировать test.id и test.run. Это работает просто.

Здесь используется особенность, известная как "пакет пространства имен" (*namespace package*). По существу, пакет пространства имен – специальный вид пакета, который используется для слияния различных каталогов кода под общим пространством имен, как показано в решении. Это может быть полезно для больших платформ, поскольку позволяет разбивать части

платформы в отдельные загрузки. Также это позволяет легко создавать сторонние дополнения и другие расширения для таких платформ.

Ключ к созданию пакета пространства имен – убедиться, что в каталоге верхнего уровня нет файлов __init__.py. Благодаря отсутствию файлов __init__.py при импорте пакета происходит интереснейшая вещь. Вместо ошибки интерпретатор начинает создавать список всех каталогов, которые, оказывается, содержат соответствующее имя пакета. Потом создается специальный модуль пакета пространства имен, а копия (доступная только для чтения) списка каталогов сохраняется в переменной __path__. Например:

```
>>> import test
>>> test.__path__
_NamespacePath(['foo-package/test', 'bar-package/test'])
>>>
```

Каталоги в __path__ используются при определении местоположения дальнейших субкомпонентов пакета (например, при импорте *test.run* или *test.id*).

Важная функция пакетов пространства имен – то, что любой может расширить пространство имен с помощью своего собственного кода. Например, представим, что вы создали собственный каталог кода:

```
my-package/
  test/
    custom.py
```

Если вы добавите ваш каталог с кодом в sys.path вместе с другими пакетами,

то он будет беспрепятственно объединен с другими каталогами пакета *test*:

```
>>> import test.custom
>>> import test.run
>>> import test.id
>>>
```

Проверить, является ли пакет пакетом пространства имен, можно посредством анализа его атрибута __file__. Если этот атрибут отсутствует, значит, пакет является пакетом пространства имен. Также строка представления будет содержать слово "namespace":

```
>>> test.__file__
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'

>>> test
<module 'test' (namespace)>
>>>
```

13.9. Перезагрузка модулей

Для перезагрузки ранее загруженного модуля используйте imp.reload(). Например:

```
>>> import test
>>> import imp
>>> imp.reload(test)
<module 'test' from './test.py'>
>>>
```

Перезагрузка модуля очень полезна при отладке и разработке, но не очень безопасна в производственном коде, поскольку не всегда работает так, как вы ожидаете.

За сценой операция *reload()* стирает содержимое базового словаря модуля и обновляет его, заново выполнив код модуля. Идентификационные данные самого объекта модуля остаются неизменными. Таким образом, эта операция обновляет модуль везде, где он был импортирован в программу.

Однако операция *reload()* не обновляет определения, которые были импортированы с использованием операторов, таких как *from module import name*. В качестве примера рассмотрим следующий код:

```
# test.py
def bar():
  print('bar')
def run():
  print('run')
```

Теперь запустим интерактивный сеанс:

```
>>> import test
```

```
>>> from test import run
>>> test.bar()
bar
>>> run()
run
>>>
```

Не выходя из Python, отредактируйте исходный код test.py, так чтобы функция *run()* была такой:

```
def run():
  print('New run')
```

Теперь вернемся в наш интерактивный сеанс, выполним перезагрузку модуля и повторим этот эксперимент:

```
>>> import imp
>>> imp.reload(test)
<module 'test' from './test.py'>
>>> test.bar()
bar
>>> run()             # Старый вывод
run
>>> test.run()        # Новый вывод
New run
>>>
```

В этом примере, как вы успели заметить, загруженный две версии функции run(). Обычно это не то, что вам нужно и в итоге простые на первый взгляд вещи приводят, в конечном счете, к головной боли.

Именно по этой причине нужно стараться избегать использования перезагрузки модулей в производственном коде. Пусть она останется уделом отладки и использования в интерактивных сеансах, где вы можете экспериментировать с кодом без выхода из интерпретатора.

13.10. Создание каталога или zip-архива, выполняемого как главный сценарий

Иногда программы разрастаются и состоят из множества файлов. А Вы бы хотели получить простой способ выполнения этой программы?

Если ваша программа состоит из множества файлов, вы можете поместить ее в отдельный каталог и создать файл __main__.py. Например, вы можете создать подобный каталог:

```
myapplication/
  test.py
  bar.py
  run.py
  __main__.py
```

Если в каталоге присутствует файл __main__.py, вы можете запустить интерпретатор просто так:

```
$ python3 myapplication
```

Интерпретатор автоматически запустит файл __main__.py как основной файл программы. Данная техника также работает, если вы запакуете весь ваш код в Zip-архив. Например:

```
$ ls
test.py bar.py run.py __main__.py
$ zip -r myapp.zip *.py
$ python3 myapp.zip
... вывод из __main__.py ...
```

Создание каталога или zip-архива и добавление файла __main__.py – один из возможных способов упаковки большого Python-приложения. Данный способ отличается от упаковки кода в пакет, поскольку код не предназначен для использования в качестве модуля стандартной библиотеки. Вместо этого данный способ позволяет создавать пакеты Python-кода, которые могут быть легко выполнены кем-то еще.

Так как каталоги и zip-архивы отличаются от обычных файлов, вы можете также добавить сценарий оболочки, чтобы упростить выполнение вашего кода. Например, если ваш код находится в архиве myapp.zip, вы можете создать следующий сценарий оболочки:

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

13.11. Добавление каталогов в sys.path

Иногда приходится работать с кодом, который не может быть импортирован Python, потому что расположен в каталоге, не перечисленным в sys.path. Вам нужно добавить новые каталоги в sys.path, но вам не хочется делать это в своем коде. Что делать?

Есть два общих способа добавить новые каталоги в sys.path. Первый заключается в использовании переменной окружения PYTHONPATH. Например:

```
$ env PYTHONPATH=/some/dir:/other/dir python3
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> sys.path
['', '/some/dir', '/other/dir', ...]
>>>
```

В пользовательском приложении эту переменную окружения можно установить при запуске программы или через сценарий оболочки.

Второй способ – создать файл .pth, который перечисляет необходимые каталоги, например:

```
# myapplication.pth
/some/dir
/other/dir
```

Этот .pth файл нужно поместить в один из каталогов site-packages, который обычно находится в /usr/local/lib/python3.9/site-packages или ~/.local/lib/python3.9/sitepackages.

При запуске интерпретатора каталоги, перечисленные в .pth-файле, будут добавлены в sys.path (если они существуют в файловой системе). Уста-

новка .pth-файла может потребовать прав администратора, если он добавляется в общесистемный каталог (/usr/local/lib/python3.9/site-packages).

Столкнувшись с подобной проблемой, первое, что приходит в голову – написать код, вручную корректирующий значение sys.path, например:

```
import sys
sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

Хотя это работает, такой подход чрезвычайно хрупкий и рекомендуется его избегать. Проблема в том, что вы явно указываете в своем коде имена каталогов. В результате рано или поздно это приведет к проблеме – когда код будет перемещен в какое-то другое расположение. Лучше сконфигурировать путь в другом месте и так, чтобы его можно было изменить, не редактируя исходный код вашей программы.

Иногда вы можете использовать подобный способ, но только в случае, если вы надлежащим образом создаете абсолютный путь, например, используя переменные уровня модуля, такие как __file__. Например:

```
import sys
from os.path import abspath, join, dirname
sys.path.insert(0, abspath(dirname('__file__'), 'src'))
```

В результате каталог **src** будет добавлен в путь корректным образом: вы не указываете абсолютный путь, а формируете его на основании переменной __file__.

Каталоги site-packages содержат сторонние модули и пакеты, установленные в вашей системе. Как правило, все сторонние модули и пакеты устанавливаются в эти пакеты. Не смотря на то, что .pth-файлы находятся в этих каталогах, они могут ссылаться на любые каталоги в системе. Таким образом, реально ваш код может находиться за пределами каталогов site-packages, пока его местоположение указано в .pth-файле.

13.12. Распространение пакетов

Вы написали полезную библиотеку, и вы хотите сделать ее доступной другим. Первым делом нужно придумать вашей библиотеке уникальное имя и

очистить структуру каталогов. Например, типичный пакет библиотеки может выглядеть примерно так:

```
projectname/
  README.txt
  Doc/
    documentation.txt
  projectname/
    __init__.py
    foo.py
    bar.py
    utils/
      __init__.py
      test.py
      run.py
  examples/
    helloworld.py
...
```

Чтобы сделать ваш пакет пригодным к распространению, напишите файл setup.py, который выглядит примерно так:

```
# setup.py
from distutils.core import setup

setup(name='projectname',
  version='1.0',
  author='Ваше имя',
  author_email='you@example.com',
  url='http://www.you.com/projectname',
  packages=['projectname', 'projectname.utils'],
)
```

Далее нужно создать файл MANIFEST.in, в котором перечислены различные файлы, не имеющие отношения к исходному коду:

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

Убедитесь, что поместили файлы setup.py и MANIFEST.in в каталог верхнего уровня для вашего пакета. Далее, чтобы создать дистрибутив пакета, введите команду:

```
$ python3 setup.py sdist
```

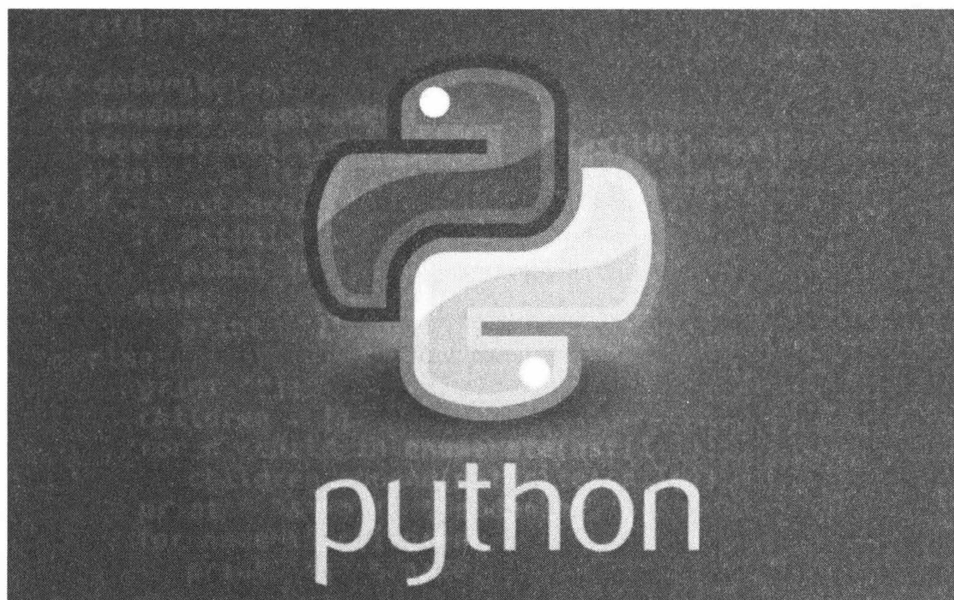
В результате будет создан файл вроде *projectname-1.0.zip* или *projectname-1.0.tar.gz* – в зависимости от платформы. Если все работает, вы можете отправить этот файл кому-то или же загрузить в индекс пакетов Python (*https://pypi.python.org/pypi*).

Для чистого кода Python написание файла setup.py – несложное занятие. Один тонкий момент в том, что вы должны вручную перечислить каждый подкаталог, хранящий файлы с исходным кодом. Часто программисты указывают только каталог верхнего уровня пакета и забывают описать подкомпоненты пакета. Это неправильно! Это – то, почему спецификация для пакетов в setup.py содержит список packages=['projectname', 'projectname.utils'].

Большинство программистов Python знает, что есть множество других (сторонних) средств создания дистрибутива пакета, в том числе ***setuptools***. Некоторые из них являются заменой библиотеки ***distutils*** и могут быть найдены в стандартной библиотеке. Знайте, что если вы полагаетесь на эти пакеты, то пользователи не смогут установить ваше программное обеспечение, если они также не установили требуемый диспетчер пакетов. Старайтесь сохранять вещи максимально простыми. Как минимум, убедитесь, что ваш код может быть установлен, используя стандартную установку Python 3. Дополнительные функции могут поддерживаться опционально, если доступны дополнительные пакеты.

ГЛАВА 14.

ОБРАБОТКА ИСКЛЮЧЕНИЙ



14.1. Что такое исключение?

Исключение – это извещение интерпретатора об ошибке в программном коде или о каком-то другом событии. Если в коде программы вы не предусмотрите обработку исключений, то выполнение программы будет прервано и будет выведено сообщение об ошибке.

Существует три типа ошибок: ***синтаксические***, ***логические*** и ***ошибки времени выполнения***. Первый тип ошибок – самый простой. Это ошибки в синтаксисе языка, как правило, интерпретатор предупреждает о наличии таких ошибок, а выполнение программы будет прервано. Простота этих ошибок в том, что интерпретатор сообщает в какой строке ошибка и вам остается лишь исправить ее.

Логические ошибки – более коварны. С точки зрения синтаксиса все нормально, но вот результаты работы программы не соответствуют ожидаемым. Понять, что в программе ошибка можно только после анализа работы программы.

Ошибки времени выполнения возникают во время выполнения программы. Причина таких ошибок – события, не предусмотренные программистом. Например, вы создали функцию деления двух чисел и не предусмотрели, что на 0 делить нельзя. В результате, если передать в качестве делителя 0,

то возникнет ошибка деления на ноль – это классическая ошибка времени выполнения:

```
>>> a = 10 / 0
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a = 10 / 0
ZeroDivisionError: division by zero
>>>
```

Если вы не предусмотрите обработку исключения *ZeroDivisionError*, то выполнение вашей программы будет прервано.

Второе частое событие – это *ValueError*. Оно может возникнуть, если подстрока не найдена:

```
>>> "Hello".index('hi')
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    "Hello".index('hi')
ValueError: substring not found
```

При работе со списками, кортежами могут возникнуть ошибки *IndexError* – когда вы пытаетесь получить доступ к несуществующему элементу списка:

```
>>> a = [1, 2, 3]
>>> a[6] = 1
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a[6] = 1
IndexError: list assignment index out of range
>>>
```

14.2. Типы исключений

Разные типы ошибок генерируют разные типы исключений. Как было показано ранее, при попытке открыть несуществующий файл, было сгенериро-

вано исключение `FileNotFoundError`, а при попытке преобразовать число в строку – `ValueError`.

Разные типы исключений представлены в таблице 14.1. Всего существует более 20 исключений, мы же рассмотрим только самые популярные.

Таблица 14.1. Самые распространенные исключения

Исключение	Описание
IOError	Генерируется, если невозможно выполнить операцию ввода/вывода
IndexError	Генерируется, если в последовательности не найден элемент с заданным индексом
KeyError	Если в словаре не найден указанный ключ
NameError	Если не найдено имя (переменной или функции)
SyntaxError	Если в коде обнаружена синтаксическая ошибка
TypeError	Если стандартная операция применяется к объекту неподходящего типа
ValueError	Если операция или функция принимает аргумент с неподходящим значением
ZeroDivisionError	Если есть деление на 0

В Python вы можете использовать следующие встроенные классы исключений:

- **BaseException** – начиная с Python 2.5, является классом самого верхнего уровня
- **Exception** – именно этот класс, а не BaseException, необходимо наследовать при создании пользовательских исключений
- **AssertionError** – возбуждается инструкцией *assert*

- **AttributeError** – попытка обращения к несуществующему атрибуту объекта
- **EOFError** – возбуждается функция input() и raw_input() при достижении конца файла
- **IOError** – ошибка доступа к файлу (ошибка ввода/вывода)
- **ImportError** – невозможно подключить модуль или пакет
- **IndentationError** – неправильно расставлены отступы в программе
- **IndexError** – указанный индекс не существует в последовательности
- **KeyError** – указанный ключ не существует в словаре
- **KeyboardInterrupt** – нажата комбинация клавиш Ctrl+C
- **NameError** – попытка обращения к идентификатору до его определения
- **StopInteration** – возбуждается метод next() как сигнал об окончании итерации
- **SyntaxError** – синтаксическая ошибка
- **TypeError** – тип объекта не соответствует ожидаемому
- **UnboundLocalError** – внутри функции переменной присваивается значение после обращения к одноименной глобальной переменной
- **UnicodeDecodeError** – ошибка преобразования обычной строки в Unicode строку
- **UnicodeEncodeError** – ошибка преобразования Unicode строки в обычную строку
- **ValueError** – переданный параметр не соответствует ожидаемому значению
- **ZeroDivisionError** – попытка деления на ноль

Иерархия классов исключений выглядит так:

```
BaseException
    GeneratorExit (в Python 2.6 и выше)
    KeyboardInterrupt
    SystemExit
    Exception
        GeneratorExit (в Python 2.5)
        StopIteration
```

```
        Warning
            BytesWarning (в Python 2.6 и выше)
            DeprecationWarning, FutureWarning, ImportWarning
            PendingDeprecationWarning, RuntimeWarning,
SyntaxWarning
            UnicodeWarning, UserWarning
        StandardError
            ArithmeticError
                FloatingPointError, OverflowError,
ZeroDivisionError
            AssertionError
            AttributeError
            BufferError (в Python 2.6)
            EnvironmentError
                IOError
                OSError
                    WindowsError
            EOFError
            ImportError
            LookupError
                IndexError, KeyError
            MemoryError
            NameError
                UnboundLocalError
            ReferenceError
            RuntimeError
                NotImplementedError
            SyntaxError
                IndentationError
                    TabError
            SystemError
            TypeError
            ValueError
                UnicodeError
                    UnicodeDecodeError, UnicodeEncodeError
                    UnicodeTranslateError
```

Мы можем заставить интерпретатор реагировать не на все исключения, а только на определенные, например:

Листинг 14.1. Пример обработки ValueError

```
try:
    k = int(input("Введите целое число: "))
    print("Вы ввели: ", k)
except ValueError:
```

```
    print("Нужно ввести целое!!!")
```

Здесь мы обрабатываем исключение определенного типа. При необходимости можно обработать несколько исключений:

Листинг 14.2. Пример обработки нескольких исключений

```
try:
    a = int(input("Введите целое a: "))
    b = int(input("Введите целое b: "))
    print("a/b = ", a/b)
except ValueError:
    print("Нужно ввести целое!!!")
except ZeroDivisionError:
    print("Деление на 0!")
```

Да, можно обрабатывать все исключения подряд, не указывая тип исключения в **except**. Но если вы будете обрабатывать только определенные исключения, то можете указать разные сообщения пользователю – так вы сделаете свою программу информативнее.

Гибкость Python в том, что он может передать в вашу программу свое ругательство, то есть сообщение об ошибки, но при этом программа не будет прервана. Это называется передачей аргумента исключения (лист. 14.3).

Листинг 14.3. Передача аргумента исключения

```
try:
    a = int(input("Введите целое a: "))
    b = int(input("Введите целое b: "))
    print("a/b = ", a/b)
except ValueError:
    print("Нужно ввести целое!!!")
except ZeroDivisionError as e:
    print("Деление на 0!")
    print("Сообщение Python:")
    print(e)
```

На рис. 14.1 показано, что кроме наших сообщений самим Python будет выведено сообщение *division by zero*.

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: C:\temp\py\14-3.py ========================
Введите целое а: 9
Введите целое b: 0
Деление на 0!
Сообщение Python:
division by zero
>>>
Ln: 10 Col: 4
```

Рис. 14.1. Сообщение об ошибке интерпретатора вместе с пользовательским сообщением

14.3. Инструкция try..except..else..finally

Инструкция *try* используется для обработки исключений. Формат инструкции следующий:

```
try:
   <Код, в котором могут возникнуть исключения>
[except [<Исключение1> [as <Объект исключения>]]:
   <Код, выполняемый при перехвате исключения 1>]
[...
[except [<ИсключениеN> [as <Объект исключения>]]:
   <Код, выполняемый при перехвате исключения N>]]
[else:
   <Код, который будет выполнен, если исключение не возникло>]
[finally:
   <Код, который будет выполнен в любом случае>]
```

Теперь рассмотрим пример обработки исключения деления на ноль:

```
try:
   a = 10 / 0
except ZeroDivisionError:
   print('ZeroDivisionError')
   a = 0
print(a)
```

В результате вместо сообщения об ошибке будет выведена строка ZeroDivisionError (можно ее и не выводить – она предназначена для нас) и выведен 0 в качестве результата.

Инструкция работает так: если в блоке **try** возникает исключение, управление передается блоку **except**. Вы можете указать несколько блоков **except**, в каждом из которых будут описаны действия, которые будут выполнены при том или ином исключении. В блоке **else** указывается код, который будет выполнен, если исключение не возникло, а в блоке **finally** указывается код, который будет выполнен в любом случае.

Вот как можно использовать блоки **else** и **finally**:

```
try:
   x = 10 / 0
except ZeroDivisionError:
   print('ZeroDivisionError')
else:
   print('Оператор Else')
finally:
   print('Оператор Finally')
```

Сначала выполните этот код как есть. Вы увидите вывод:

```
ZeroDivisionError
Оператор Finally
```

Затем замените 0 на 5:

```
x = 10 / 5
```

Вы увидите другой вывод:

```
Оператор Else
Оператор Finally
```

14.4. Инструкция with .. as

Язык Python поддерживает протокол менеджеров контекста. Данный протокол гарантирует выполнение завершающих действий (закрытие файлов, сокетов) вне зависимости от того, произошло ли исключение внутри блока кода или нет.

Данный менеджер удобно использовать в следующем случае. Представим, что вы пишете код, который выполняет исключительную блокировку файла. Где-то в середине этого кода происходит исключение. В результате выполнение сценария будет прервано, а файл так и останется заблокированным. Менеджер контекста позволяет в любом случае надлежащим образом закрыть файлы и сетевые соединения.

Для работы с менеджером контекста как раз и используется инструкция *with..as*. Формат ее следующий:

```
with <Выражение1> [as <Переменная>][, …,
     <ВыражениеN> [as <Переменная>]]:
     <Код, в котором перехватываются исключения>
```

Сначала вычисляется выражение 1, которое должно возвращать объект, который поддерживает протокол. Данный объект должен иметь два метода: __enter__() и __exit__(). Метод __enter__() вызывается после создания объекта. Значение, возвращаемое этим методом, присваивается переменной, указанной после ключевого слова **as**. Если переменная не указана, возвращаемое значение игнорируется. Формат метода __center__(): __enter__ (self).

После этого выполняются инструкции внутри тела инструкции *with*. Если при выполнении возникло исключение, то управление будет передано методу __exit__(). Метод имеет следующий формат:

```
__exit__(self, <тип исключения>, <значение>, <объект traceback>)
```

Значения, доступные через эти три параметра, полностью эквивалентны значениям, которые возвращаются функцией exc_info() из модуля **sys**. Если исключение обработано, метод должен вернуть *True*, в противном случае – *False*. Если при выполнении операторов, расположенных внутри **with**, исключение не возникло, управление передается методу __exit__(). В этом случае три параметра будут содержать значения *None*.

Пример:

```
class SampleClass:
   def __enter__(self):
      print('Enter')
      return self
   def __exit__(self, Type, Value, Trace):
      print('Exit')
      if Type is None:      # Исключения не было
         print('Нет исключений')
      else:
         print('Значение = ', Value)
         return False # False - исключение не обработано

with SampleClass():
   print('Внутри')
   raise TypeError('TypeError Exception')
```

Последний оператор генерирует исключение вручную, чтобы проверить, работает ли наш объект или нет.

Некоторые встроенные объекты, например, файлы, по умолчанию поддерживают протокол. Вот пример работы с файлом:

```
with open('log.txt', 'a') as f:
   f.write('Error')
```

14.5. Генерирование исключений

Программист может сам сгенерировать исключение – или пользовательское или встроенное, например, как ранее было показано, мы генерировали исключение *TypeError*. Для этого используется инструкция *raise*:

```
raise <Экземпляр класса>
raise <Название класса>
raise <Экземпляр или название класса> from <Объект исключения>
raise
```

Пример:

```
>>> raise ValueError("Info")
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    raise ValueError("Info")
ValueError: Info
>>>
```

А вот как можно обработать это исключение:

```
try:
      raise ValueError("Info")
except ValueError as msg:
      print(msg)          #Выведет: Info
```

Кроме *raise* есть еще и инструкция *assert*, которая аналогична следующему коду:

```
if __debug__
      if not <логическое выражение>:
      raise AssertionError(<Инфо>)
```

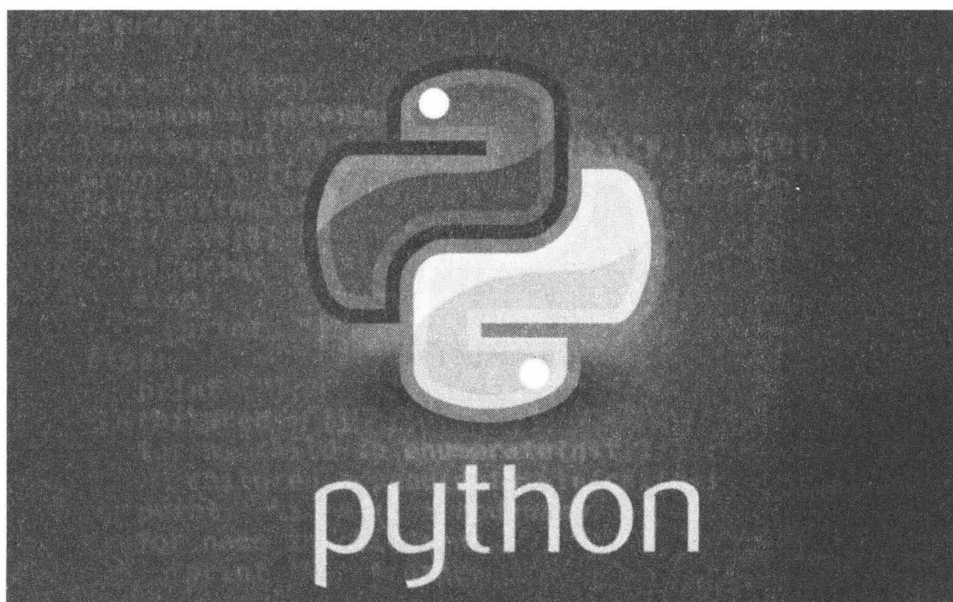
Пример:

```
try:
      x = -5
      assert x >= 0, "Error"
except AssertionError as err:
      print(err)    # Выведет Error
```

На этом все и можно перейти к чтению следующей главы, в которой будет рассмотрен файловый ввод/вывод.

ГЛАВА 15.

ФАЙЛОВЫЙ ВВОД/ВЫВОД



До этого момента мы хранили обрабатываемые данные только в переменных. Переменные – это хорошо, но они уничтожаются при завершении работы программы, а данные часто нужно хранить постоянно – даже если программа завершена или компьютер перезагружен.

15.1. Работа с файлами

15.1.1. ОТКРЫТИЕ ФАЙЛА

Открыть файл можно с помощью функции *open()*, которая имеет следующий формат:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
newline=None, closefd=True, opener=None)
```

Первый параметр задает путь к файлу. Его можно указывать, как относительный, так и абсолютный. При указании абсолютного пути в Windows

учитывайте, что символ \ является специальным и его нужно экранировать, например:

```
"C:\\Python\\file.txt"                    # Правильно
"C:\Python\file.txt"                      # Неправильно
```

Если вы не заэкранируете слэши, то получите ошибку *IOError*.

```
>>> f = "C:\Python\file.txt"
>>> open(f)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    open(f)
OSError: [Errno 22] Invalid argument: 'C:\\Python\x0cile.txt'
>>>
```

Преобразовать относительный путь в абсолютный можно с помощью функции *abspath()* из модуля os.path. Пример:

```
>>> os.path.abspath("include\\ast.h")
'c:\\Python39\\Lib\\idlelib\\include\\ast.h'
>>> os.path.abspath("ast.h")
'c:\\Python39\\Lib\\idlelib\\ast.h'
>>>
```

Я все же рекомендую использовать абсолютный путь, а не относительный. Как видите, функция *abspath()* не всегда возвращает то, что нужно. В моем случае абсолютный путь к файлу выглядит так:

```
c:\Python39\include\ast.h
```

Однако функция *abspath()* попросту добавляет к указанному вами имени (пути) текущий путь, по которому выполняется программа (в данном случае IDLE). В некоторых случаях – это то, что вам нужно. В некоторых – нет. Поэтому не ленитесь и используйте абсолютный путь вместо относительного.

После имени файла указывается режим его открытия:

- **'r'** – только чтение (по умолчанию);
- **'w'** – запись, если файл существует, он будет перезаписан;
- **'x'** – эксклюзивный доступ, если файл существует, у вас ничего не выйдет;
- **'a'** – дозапись в конец файла, удобно использовать, если файл уже существует. Если файл не существует, он будет создан;
- **'b'** – двоичный режим;
- **'t'** – текстовый режим (по умолчанию);
- **'+'** – открывает файл для обновления (чтение и запись);
- **'U'** – универсальный режим, уже не используется.

Данные режимы можно комбинировать. Рассмотрим пример открытия файла в бинарном режиме для чтения:

```
with open("somefile.txt", "rb") as f:
    for line in f:
        print(repr(line))
```

Имя файла и режим открытия – основные параметры. Остальные используются довольно редко, но мы все же рассмотрим некоторые из них. Параметр **buffering** управляет политикой буферизации. Передайте в него 0, чтобы выключить буферизацию вообще (используется только в двоичном режиме). Значение 1 включает буферизацию (доступно только в текстовом режиме). Другое положительное число задает примерный размер буфера, а отрицательное значение (или отсутствие значения) означает установку размера, применяемого в системе по умолчанию.

Параметр **errors** – это необязательный параметр, указывающий, как будут обрабатываться ошибки кодирования и декодирования в двоичном (бинарном) режиме. Если вам лень обрабатывать исключения *ValueError*, укажите значение 'ignore' для игнорирования ошибок. Значение 'replace' заставит функцию заменять каждый неизвестный символ знаком вопрос ?.

Параметр **newline** позволяет указать символ новой строки и применяется только к текстовому режиму.

Очень полезен параметр **encoding**, который позволяет задать кодировку информации в файле, например:

```
with open("somefile.txt", "a", encoding="utf-8") as f:
   f.write("nit.center")

with open("somefile.txt", "r", encoding="utf-8") as f:
   for line in f:
      print(repl(line))
```

Остальные параметры вы вряд ли будете использовать, а если вы заинтересовались, зачем они используются, вы можете прочитать о них в руководстве по функции *open()*:

https://docs.python.org/3/library/functions.html#open

15.1.2. МЕТОДЫ ДЛЯ РАБОТЫ С ФАЙЛАМИ

После того, как файл открыт, вы можете использовать методы файлового объекта для работы с файлом:

- **close()** – закрывает файл. Интерпретатор автоматически закрывает файл при завершении программы, но явное закрытие файла считается хорошим тоном
- **write(<данные>)** – записывает данные (строку или последовательность байтов) в файл
- **writelines(<последовательность>)** – записывает последовательность в файл. Если все элементы последовательности – строки, то файл можно открыть в текстовом режиме, в противном случае файл нужно открывать в бинарном режиме
- **read(<количество байтов>)** – читает указанное количество байтов из файла. Если количество не указано, то возвращается содержимое файла от текущей позиции указателя до конца файла
- **readline(<количество>)** – считывает одну строку из файла при каждом вызове. Если файл открыт в текстовом режиме, возвращается строка, если в бинарном – последовательность байтов. Вместо этого метода можно использовать его "служебную" версию __next__(), поскольку файловый объект поддерживает механизм итерации
- **flush()** – записывает данные из буфера на диск
- **fileno()** – возвращает целочисленный дескриптор файла

- **truncate(<количество>)** – обрезает файл до указанного количества символов
- **tell()** – возвращает позицию указателя относительно начала файла в виде целого числа. В Windows метод *tell()* считает символ \r дополнительным байтом, хотя этот символ и удаляется при открытии файла в текстовом режиме
- **seek(<смещение>[, <Начало>])** – устанавливает указатель в позицию <Смещение> относительно позиции <Начало>. В параметре <Начало> вы можете указать следующие атрибуты из модуля **io**:
- io.SEEK_SET (0) – начало файла (по умолчанию)
- io.SEEK_CUR (1) – текущая позиция указателя
- io.SEEK_END (2) – конец файла

Теперь рассмотрим эти методы подробнее. Начнем с *close()*. Python поддерживает протоколов менеджеров контента, гарантирующий закрытие файла вне зависимости от того, произошло исключение или нет. Пример:

```
with open("somefile.txt") as f:
   f.read()
# А здесь файл уже закрыт
```

Далее рассмотрим пример записи в файл в текстовом и двоичном режимах:

```
# Текстовый режим
f = open("somefile.txt", "w", encoding="utf-8")
f.write("String")
f.close()

# Двоичный режим
f = open("somefile.txt", "wb", encoding="utf-8")
f.write("String\r\n")
f.close()

# Запись нескольких строк
f = open("somefile.txt", "wb", encoding="utf-8")
f.writelines(["String1\n", "String2"])
f.flush()            # Сбрасываем буфер на диск
f.close()
```

Пример чтения из файла:

```
with open("somefile.txt", "r") as f:
   f.read()

# Побайтное чтение
f = open("file.txt", "r")
f.read(10)          # Читаем 10 байтов
f.read(10)          # Читаем следующие 10 байтов

print("**Посимвольное чтение с кодировкой")
txt = open("text.txt", "r", encoding='utf-8')
print(txt.read(1))
print(txt.read(2))
print(txt.read(6))
txt.close()

print("**Читает весь файл в переменную content ")
txt = open("text.txt", "r", encoding='utf-8')
content = txt.read()
print(content)
txt.close()

print("**Построчное чтение файла ")
txt = open("text.txt", "r", encoding='utf-8')
print(txt.readline())       # Строка 1
print(txt.readline())       # Строка 2
txt.close()

print("** Читает файл в список ")
txt = open("text.txt", "r", encoding='utf-8')
lines = txt.readlines()

print(lines)
print(len(lines))
for line in lines:
    print(line)

txt.close()

print("** Построчное чтение всего файла:")
txt = open("text.txt", "r", encoding='utf-8')
for line in txt:
      print(line)
txt.close()
```

Мы только что рассмотрели, как читать информацию из файла, как записывать ее в файл посредством высокоуровневых методов файлового объекта. В большинстве случаев этого достаточно в 90% случаев. Если вам нужны низкоуровневые функции для работы с файлами, вы найдете их в модуле **os**:

https://docs.python.org/3.1/library/os.html

15.1.3. ФУНКЦИИ ДЛЯ МАНИПУЛИРОВАНИЯ ФАЙЛАМИ

Кроме методов записи и чтения файла в Python есть функции манипулирования целыми файлами – копирования, удаления, перемещения файлов и т.д. Все рассматриваемые в этом разделе функции копирования и перемещения находятся в модуле **shutil**.

Функция *copyfile()* позволяет скопировать содержимое файла в другой файл. Речь идет именно о содержимом, никакие метаданные вроде прав доступа при этом не копируются. Если файл-назначение существует, он будет перезаписан. Если файл скопировать не удалось, вы получите исключение *IOError*. Синтаксис функции:

```
copyfile(<Источник>, <Назначение>)
```

Пример:

```
import shutil
shutil.copyfile("file1.txt", "file2.txt")
```

Если нужно скопировать также права доступа, то нужно использовать функцию *copy()*. Поведение функции такое же, как и у *copyfile()*. Синтаксис тоже такой же:

```
copy(<Источник>, <Назначение>)
```

Функция *copy2()* копирует не только права доступа, но и остальные метаданные:

```
copy2(<Источник>, <Назначение>)
```

Для перемещения файлов используется функция move():

```
move(<Источник>, <Назначение>)
```

Копирует файл в <Назначение>, а затем удаляет его. Если файл существует, то он будет перезаписан. В случае ошибки возбуждается исключение *IOError*. Если файл нельзя удалить именно в Windows, то генерируется исключение *WindowsError* (это исключение генерируется только в Windows, в других ОС генерируется только одно исключение – *WindowsError*).

Пример:

```
import shutil
print("Перемещение файла...")
try:
       shutil.move("file1.txt", "file2.txt")
except WindowsError:
       print("Ошибка при перемещении файла!")
else:
       print("OK")
```

Далее мы будем рассматривать функции из модуля **os**. Для переименования файла используется функция rename():

```
rename(<Старое имя>, <Новое имя>)
```

Если исходный файл отсутствует или новое имя уже существует, то в Windows будет возбуждено исключение *WindowsError*. Но поскольку *WindowsError* наследует *OSError*, то правильнее обрабатывать исключение *OSError*.

```
import os
try:
   os.rename('f1.doc', 'f2.doc')
except OSError:
   print('Ошибка!')
else:
```

```
  print('OK')
```

Для удаления файла используются функции *remove()* и *unlink()*:

```
remove(<имя файла>)
unlink(<имя файла>)
```

В случае ошибки обе функции генерируют исключение *OSError*.

В модуле os.path находятся следующие полезные функции:

- **exists(<имя>)** – проверяет существование файла и возвращает *True*, если файл существует
- **getsize(<имя>)** – возвращает размер файла. Перед вызовом этой функции желательно проверить файл на существование
- **getatime(<имя>)** – возвращает время последнего доступа файла. Возвращается так называемая метка времени (timestamp) – количество секунд, прошедших с 1 января 1970 года
- **getmtime(<имя>)** – возвращает время последнего изменения
- **getctime(<имя>)** – возвращает время создания файла

Пример:

```
import os.path
import time as t
mod_time = os.path.getmtime("hello.py")
print(t.strftime("%d.%m.%Y %H:%M:%S", t.localtime(mod_time))
```

Ранее вы уже были знакомы с одной функцией из модуля os.path – *abspath()*. Кроме этой функции в этом модуле есть и другие функции преобразования пути к файлу. Например, функция isabs(<путь>) возвращает *True*, если указанный путь является абсолютным. А функция basename(<Путь>) возвращает базовое имя файла, то есть имя файла без пути к нему. А функция dirname(<путь>), наоборот, возвращает путь без базового имени. Идентификатор *sep* содержит разделитель элементов пути, используемый в вашей операционной системе. Примеры:

```
>>> from os.path import *
>>> sep
```

```
'\\'
>>> basename("c:\temp\hello.py")
'hello.py'
>>> dirname("C:\temp\hello.py")
'C:\temp'
```

Обратите внимание: даже не смотря на то, что я неправильно указал разделитель пути (поскольку я использую Windows, то я должен был использовать разделитель \\, как показано выше), функции os.path справились со своей задачей без ошибок и правильно определили базовое имя и название каталога, в котором находится файл.

15.2. Работа с каталогами

В модуле **os** находятся также и функции для работы с каталогами. Начнем с функции *getcwd()*, которая возвращает текущий рабочий каталог:

```
>>> from os import *
>>> getcwd()
'C:\\Python39\\Lib\\idlelib'
>>>
```

Функция *chdir()* изменяет текущий рабочий каталог:

```
>>> chdir('c:\\test')
>>> getcwd()
'c:\\test'
>>>
```

Для создания каталога нужно использовать функцию mkdir(<имя каталога>[, <права доступа>]). Второй параметр не обязательный, он задает права доступа в UNIX/Linux. Если вы хотите его использовать, то нужно передать трехзначное число в восьмеричной системе, например, 0o777. О правах доступа мы поговорим в следующем разделе. Пример:

```
>>> mkdir('dir1')
```

Удалить пустой каталог можно с помощью функции rmdir(<имя каталога>). Обратите внимание: каталог должен быть пустым, иначе вы получите исключение *OSError*.

Вывести содержимое каталога позволяет функция *listdir()*:

```
>>> listdir(getcwd())
['dir1', 'log2']
>>>
```

Обойти дерево каталогов можно с помощью функции *walk()*. Такая функция стандартна для Python и вам не придется изобретать велосипед заново. Формат функции:

```
walk(<Начальный каталог>[, topdown=True]
     [, onerror=None][, followlinks=False])
```

В качестве значения функция *walk()* возвращает объект, на каждой итерации которого доступен кортеж из трех элементов – текущий каталог, список файлов и список каталогов. Параметр **topdown** задает порядок обхода каталога – прямой или обратный.

Пример:

```
>>> for (a, b, c) in walk('c:\\test'): print(a)

c:\test
c:\test\dir1
>>> for (a, b, c) in walk('c:\\test', False): print(a)

c:\test\dir1
c:\test
>>>
```

При обходе дерева каталогов полезно знать, какой перед нами элемент – файл или каталог. Функция *isdir()* возвращает *True*, если обрабатываемый элемент – каталог. Аналогично, функция *isfile()* возвращает *True*, если обрабатываемый элемент – файл:

```
>>> isdir('c:\\test\\dir1')
True
>>> isfile('c:\\test\\dir1')
False
>>>
```

Обе функции находятся в модуле os.path.

Удалить дерево каталогов можно функцией *rmtree()* из shutil:

```
import shutil
rmtree("c:\\test")
```

15.3. Работа с файлами в разных форматах

15.3.1. РАБОТА С CSV

Данная глава концентрируется на использовании Python для обработки данных, представленных в разных форматах, например, в CSV, JSON, XML и двоичных упакованных записях. Мы не будем рассматривать различные алгоритмы по обработке данных, но вместо этого мы рассмотрим способы ввода и вывода данных в программу и из нее.

Формат CSV (*Comma-Separated Values*) используется для хранения электронных таблиц. Например, вы можете экспортировать электронную таблицу Excel в этот формат. Для большинства видов CSV-данных используйте библиотеку **csv**.

Представим, что у нас есть некоторая электронная таблица в файле table.csv. Вот код, который может обработать этот файл:

```
import csv

with open('table.csv') as f:
  f_csv = csv.reader(f)
  headers = next(f_csv)
  for row in f_csv:
    # Обрабатываем строку
```

```
    ...
```

В предыдущем коде ряд будет кортежем. Поэтому для доступа к определенным полям вам нужно использовать индексирование, например, row[0] (поле Symbol) и row[4] (поле Change).

Поскольку индексирование часто запутывает программиста, рассмотрите использование именованных кортежей. Например:

```
from collections import namedtuple

with open('table.csv') as f:
  f_csv = csv.reader(f)
  headings = next(f_csv)
  Row = namedtuple('Row', headings)
  for r in f_csv:
    row = Row(*r)
    # Обрабатываем строку
    ...
```

В итоге вы можете использовать заголовки колонок вроде row.FirstName и row.LastName вместо индексов. Конечно, это только в том случае, если заголовки колонок являются допустимыми идентификаторами Python. Если это не так, обращаться к данным по заголовкам будет не очень удобно.

Другая альтернатива – чтение данных как последовательности словарей. Чтобы сделать это, используйте этот код:

```
import csv
with open('table.csv') as f:
  f_csv = csv.DictReader(f)
  for row in f_csv:
  # обрабатываем строку
  ...
```

В этой версии получить доступ к элементам каждого ряда тоже можно с использованием заголовков. Например: row['FirstName'] или row['LastName'].

Чтобы записать CSV-данные, вы также можете использовать модуль **csv**, но создайте объект **writer**. Например:

```
headers = ['UserID','FirstName','LastName']
rows = [('user1', 'John', 'Doe'),
```

```
('user2', 'Jane', 'Doe')
]

with open('users.csv','w') as f:
  f_csv = csv.writer(f)
  f_csv.writerow(headers)
  f_csv.writerows(rows)
```

По умолчанию библиотека **csv** запрограммирована, чтобы понимать правила кодирования CSV, используемые Microsoft Excel. Это наиболее распространенный вариант и предоставит вам лучшую совместимость. Однако если вы обратитесь к документации по **csv**, вы обнаружите, что есть возможность подстроить кодирование под разные форматы (например, изменить символ разделителя). Например, если вместо разделителя нужно использовать символ табуляции, используйте следующий код:

```
with open('users.csv') as f:
  f_tsv = csv.reader(f, delimiter='\t')
  for row in f_tsv:
  # Обрабатываем строку
...
```

15.3.2. ЧТЕНИЕ И ЗАПИСЬ JSON-ДАННЫХ

Модуль **json** предоставляет простой способ кодировать и декодировать данные в JSON. Две основные функции – *json.dumps()* и *json.loads()*, их работа похожа на других функций сериализации в других библиотеках, например, в pickle.

JSON-кодирование поддерживает базовые типы данных – *None*, *bool*, *int*, *float* и *str*, а также списки, кортежи и словари, состоящие из тех базовых типов. Для словарей считается, что ключами будут строки (любые нестроковые ключи в словаре конвертируются в строки при кодировании). Чтобы быть совместимыми со спецификацией JSON, вы должны кодировать только списки и словари. Кроме того, в веб-приложениях принято, что объект верхнего уровня является словарем.

Формат кодирования JSON почти идентичен синтаксису Python за исключением нескольких незначительных изменений. Например, *True* отображается в *true*, *False* – в *false*, а *None* – в *null*.

Вот как преобразовать структуру данных Python в JSON:

```
import json
```

```
data = {
  'firstname' : 'John',
  'lastname' : 'Doe',
  'year' : 1979
}
json_str = json.dumps(data)
```

А вот как вы можете преобразовать JSON-закодированную строку обратно в структуру данных Python:

```
data = json.loads(json_str)
```

Если вы работаете с файлами, а не строками, вы можете использовать функции *json.dump()* и *json.load()* для кодирования и декодирования JSON-данных. Например:

```
# Записываем JSON-данные
with open('data.json', 'w') as f:
json.dump(data, f)
# Читаем данные обратно
with open('data.json', 'r') as f:
data = json.load(f)
```

15.3.3. ПАРСИНГ XML-ФАЙЛОВ

Для извлечения данных из простого XML-документа можно использовать модуль xml.etree.ElementTree. Чтобы проиллюстрировать, предположим, что вы хотите проанализировать и сделать сводку произвольной RSS-ленты, содержащей элементы **title**, **pubDate**, **link**. Вот сценарий, который делает это:

```
from urllib.request import urlopen
from xml.etree.ElementTree import parse

# Загружаем RSS-ленту и парсим ее
u = urlopen('http://сайт/rss20.xml')
doc = parse(u)
# Извлекаем и выводим интересующие теги
for item in doc.iterfind('channel/item'):
```

```
  title = item.findtext('title')
  date = item.findtext('pubDate')
  link = item.findtext('link')

  print(title)
  print(date)
  print(link)
  print()
```

Очевидно, если вы хотите произвести дополнительную обработку, вам нужно заменить операторы *print()* на что-то более интересное.

Работа с данными, представленными в виде XML, осуществляется во многих приложениях. Мало того, что XML широко используется в качестве формата для обмена данными в Интернете, также он является стандартным форматом для хранения данных приложения (например, при обработке текста, в музыкальных библиотеках и т.д.). Все сказанное далее предполагает, что читатель уже знаком с основами XML.

Во многих случаях, когда XML используется просто для хранения данных, структура документа компактна и понятна. Например, вот типичная RSS-лента:

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/
elements/1.1/">
<channel>
  <title>RSS feed</title>
  <link>http://сайт/</link>
  <language>ru</language>
  <description>Описание</description>
  <item>
    <title>Заголовок 1</title>
    <link>Ссылка 1</link>
    <description>Инфо 1</description>
    <pubDate>Дата 1</pubDate>
  </item>
  <item>
    <title>Заголовок 2</title>
    <link>Ссылка 2</link>
    <description>Инфо 2</description>
    <pubDate>Дата 2</pubDate>
  </item>
...
```

```
</channel>
</rss>
```

Функция xml.etree.ElementTree.parse() парсит весь XML-документ в объект документа. Вы можете использовать методы вроде *find()*, *iterfind()* и *findtext()* для поиска определенных XML-документов. Аргументы к этим функциям – имена определенных тегов, вроде **channel/item** или **title**.

При определении тегов вы должны принять полную структуру документа во внимание. Каждая операция **find** работает относительно начального элемента. Аналогично, имя тега, которое вы передаете каждой операции, тоже указывается относительно начального элемента. В примере вызов к doc.iterfind('channel/item') находит все элементы "item", которые находятся внутри элемента "channel". "doc" представляет верхнюю часть документа (элемент "rss"). Более поздние вызовы *item.findtext()* будут иметь место относительно найденных элементов "item".

У каждого элемента, представленного модулем ElementTree, есть несколько существенных атрибутов и методов, которые полезны при парсинге. Атрибут **tag** содержит имя тега, атрибут **text** содержит текст, а метод *get()* может быть использоваться для извлечения атрибутов.

Нужно отметить, что xml.etree.ElementTree – не единственное средство для парсинга XML. Для более сложных приложений вы можете рассмотреть использование lxml[1]. Эта библиотека использует тот же API, что и ElementTree, таким образом, пример, показанный только что будет работать и с lxml. Просто нужно заменить первый оператор импорта на from lxml.etree import parse.lxml. Библиотека lxml лучше совместима с XML-стандартами. Также она быстрее и предоставляет дополнительные функции вроде валидации, XSLT и XPath.

15.3.4. ПРЕОБРАЗОВАНИЕ СЛОВАРЯ В XML

Иногда нужно сохранить содержимое словаря в XML-формате. Хотя би-

блиотека xml.etree.ElementTree обычно используется для парсинга, она также может быть использована для создания XML-документов. Например, рассмотрим эту функцию:

```
from xml.etree.ElementTree import Element
```

1 https://pypi.python.org/pypi/lxml

```
def dict_to_xml(tag, d):
    '''
    Преобразуем простой словарь из пар  ключей/значений в XML
    '''
    elem = Element(tag)
    for key, val in d.items():
        child = Element(key)
        child.text = str(val)
        elem.append(child)
    return elem
```

Вот пример использования:

```
>>> s = { 'name': 'Mark', 'shares': 70, 'price':590.1 }
>>> e = dict_to_xml('test', s)
>>> e
<Element 'test' at 0x1004b64c8>
>>>
```

Результат этого преобразования – экземпляр Element. Для ввода/вывода проще конвертировать это в байтовую строку, используя функцию *tostring()* в xml.etree.ElementTree. Например:

```
>>> from xml.etree.ElementTree import tostring
>>> tostring(e)
b'<test><price>590.1</price><shares>70</shares><name>Mark</
name></test>'
>>>
```

Если вы хотите присоединить атрибуты к элементу, используйте метод *set()*:

```
>>> e.set('_id','1234')
>>> tostring(e)
b'<test _id="1234"><price>590.1</price><shares>70</
shares><name>Mark</name>
</test>'
>>>
```

При создании XML программисты обычно просто создают XML-строку. Например:

```
def dict_to_xml_str(tag, d):
  '''
  Аналог нашей функции, но здесь просто создается XML-строка
  '''
  parts = ['<{}>'.format(tag)]
  for key, val in d.items():
    parts.append('<{0}>{1}</{0}>'.format(key,val))
  parts.append('</{}>'.format(tag))
  return ''.join(parts)
```

Проблема заключается в том, что вы можете запутаться, если попытаетесь заняться обработкой XML-файла вручную. Например, что произойдет, если значения словаря содержат специальные символы?

```
>>> d = { 'name' : '<spam>' }

>>> # Создание строки
>>> dict_to_xml_str('item',d)
'<item><name><spam></name></item>'
>>> # Правильное создание XML
>>> e = dict_to_xml('item',d)
>>> tostring(e)
b'<item><name>&lt;spam&gt;</name></item>'
>>>
```

Обратите внимание, как в последнем примере заменяются символы < и > на < и >.

Для справки: если вы даже вам захочется вручную обрабатывать такие специальные символы, вы можете использовать функции *escape()* и *unescape()* в xml.sax.saxutils. Например:

```
>>> from xml.sax.saxutils import escape, unescape
>>> escape('<spam>')
'&lt;spam&gt;'
>>> unescape(_)
'<spam>'
>>>
```

Кроме создания корректного вывода есть и другая причина, почему лучше создавать экземпляры Element вместо строк – они могут быть легко объединены вместе, чтобы создать большой документ. Результирующие экземпляры Element также могут быть обработаны разными способами и при этом не

нужно волноваться о парсинге XML. По существу, вы можете сделать всю обработку данных в более высокоуровневой форме, а затем вывести его как строку в самом конце.

15.3.5. МОДИФИКАЦИЯ И ПЕРЕЗАПИСЬ XML-КОДА

Другая часто распространенная задача – нужно прочитать XML-документ, внести в него изменения и записать обратно как XML.

Модуль xml.etree.ElementTree упрощает выполнение таких задач. По существу, вы начинаете парсинг документа обычным способом. Например, предположим, что у вас есть документ, который называется pred.xml и похож на это:

```
<?xml version="1.0"?>
<stop>
  <id>14791</id>
  <nm>Моя компания</nm>
  <sri>
    <rt>22</rt>
    <d>Улица</d>
    <dd>Индекс</dd>
  </sri>
  <cr>22</cr>
  <pre>
    <pt>5000</pt>
    <fd>Test</fd>
    <v>1378</v>
    <rn>22</rn>
  </pre>
  <pre>
    <pt>10000</pt>
    <fd>Test</fd>
    <v>1867</v>
    <rn>22</rn>
  </pre>
</stop>
```

Далее приведен пример использования ElementTree для чтения этого файла и внесения изменения в его структуру:

```
>>> from xml.etree.ElementTree import parse, Element
```

```
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>
>>> # Удаляем несколько элементов
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))
>>> # Вставляем новый элемент после <nm>...</nm>
>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
>>> e.text = 'Test'
>>> root.insert(2, e)
>>> # Записываем результат обратно в файл
>>> doc.write('newpred.xml', xml_declaration=True)
>>>
```

В результате этих операций будет создан новый XML-файл, который выглядит так:

```
<?xml version='1.0' encoding='us-ascii'?>
<stop>
  <id>14791</id>
  <nm>Моя компания</nm>
  <spam>Тест</spam><pre>
    <pt>5000</pt>
    <fd>Тест</fd>
    <v>1378</v>
    <rn>22</rn>
  </pre>
  <pre>
    <pt>1000</pt>
    <fd>Тест</fd>
    <v>1867</v>
    <rn>22</rn>
  </pre>
</stop>
```

Изменение структуры XML-документа достаточно простое, но вы должны помнить, что все модификации обычно делаются в родительском элементе, как будто это список. Например, если вы удаляете элемент, то он удаляется из его непосредственного родителя, используя метод *remove()* родителя. Если вы вставляете или добавляете новые элементы, то вы также использу-

ете методы *insert()* или *append()* родителя. Элементами можно также управлять, используя индексы и слайсы, например, element[i] или element[i:j].

Если вам нужно создать новые элементы, используйте класс Element, как было показано.

15.3.6. ДЕКОДИРОВАНИЕ И КОДИРОВАНИЕ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

Вам нужно декодировать строку шестнадцатеричных чисел в байтовую строку, или же кодировать байтовую строку как шестнадцатеричную. Для этого можно использовать модуль **binascii**. Например:

```
>>> # Начальная байтовая строка
>>> s = b'hello'
>>> # Кодируем в шестнадцатеричном виде
>>> import binascii
>>> h = binascii.b2a_hex(s)
>>> h
b'68656c6c6f'
>>> # Декодируем в байты
>>> binascii.a2b_hex(h)
b'hello'
>>>
```

Подобная функциональность также может быть найдена в модуле **base64**. Например:

```
>>> import base64
>>> h = base64.b16encode(s)
>>> h
b'68656C6C6F'
>>> base64.b16decode(h)
b'hello'
>>>
```

По большей части преобразование в шестнадцатеричную форму и обратно с использованием описанных функций очень простое. Основное различие между этими двумя методами в обработке регистра. Функции *base64*.

b16decode() и *base64.b16encode()* работают только с прописными шестнадцатеричными буквами, тогда как функции в **binascii** с любым регистром.

Важно отметить, что вывод, произведенный функциями кодирования, всегда является байтовой строкой. Чтобы привести его к Unicode, нужно добавить дополнительный шаг декодирования. Например:

```
>>> h = base64.b16encode(s)
>>> print(h)
b'68656C6C6F'
>>> print(h.decode('ascii'))
68656C6C6F
>>>
```

При декодировании шестнадцатеричных чисел функции *b16decode()* и *a2b_hex()* принимают или байты или Unicode-строки. Однако эти строки должны содержать ASCII-кодированные шестнадцатеричные цифры.

15.3.7. КОДИРОВАНИЕ/ДЕКОДИРОВАНИЕ BASE64

Кодирование Base64 используется на байтовых данных, такие как байтовые строки и массивы байтов. Кроме того, результат кодирования всегда является байтовой строкой. Если вы смешиваете данные, закодированные в Base64, с текстом Unicode, вам придется выполнить дополнительный шаг декодирования.

В модуле **base64** есть две функции – *b64encode()* и *b64decode()*, которые позволяют работать с Base64-кодированием. Например:

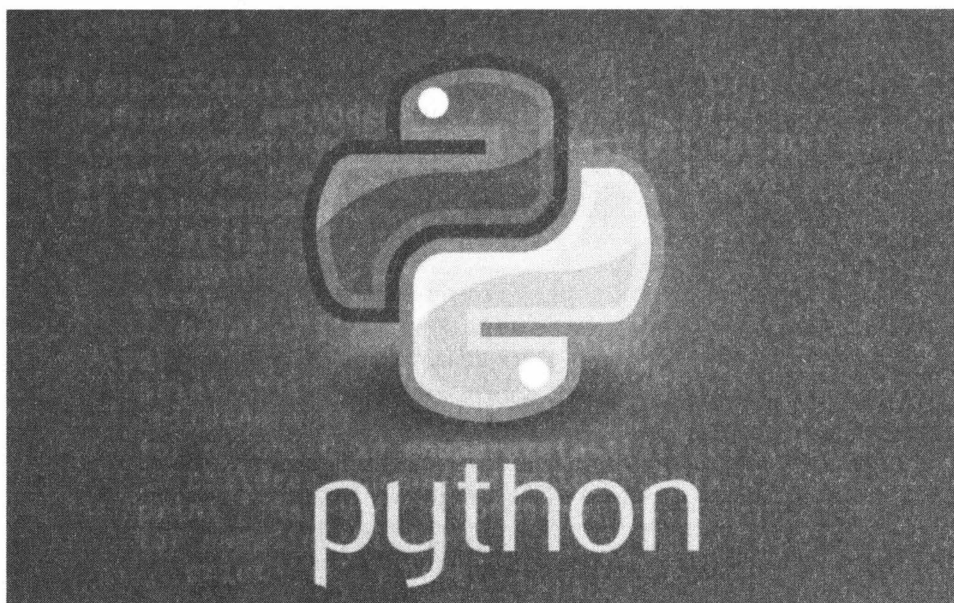
```
>>> # Некоторые байтовые данные
>>> s = b'hello'
>>> import base64

>>> # Кодируем как Base64
>>> a = base64.b64encode(s)
>>> a
b'aGVsbG8='

>>> # Декодируем данные из Base64
>>> base64.b64decode(a)
b'hello'
>>>
```

ГЛАВА 16.

ООП И PYTHON



16.1. Основы объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — это особый подход к написанию программ. Чтобы понять, что такое ООП и зачем оно нужно, необходимо вспомнить некоторые факты из истории развития вычислительной техники. Первые программы вносились в компьютер с помощью переключателей на передней панели компьютера – в то время компьютеры занимали целые комнаты. Такой способ "написания" программы, сами понимаете, был не очень эффективным – ведь большая часть времени (несколько часов, иногда – целый рабочий день) занимало подключение кабелей и установка переключателей. А сами расчеты занимали считанные минуты. Вы только представьте, что делать, если один из программистов (такие компьютеры программировались, как правило, группами программистов) неправильно подключил кабель или установил переключатель? Да, приходилось все перепроверять – по сути, все начинать заново.

Позже появились перфокарты. Программа, то есть последовательность действий, которые должен был выполнен компьютер, наносилась на перфокарту. Пользователь вычислительной машины (так правильно было на-

зывать компьютеры в то время) писали программу, оператор "записывал" программу на перфокарту, которая передавалась оператору вычислительного отдела. Через определенное время оператор возвращал пользователю результат работы программы – рулон бумаги с результатами вычислений. Мониторов тогда не было, а все, что выводил компьютер, печаталось на бумаге. Понятно, если в расчетах была допущена ошибка (со стороны пользователя, компьютеры ведь не ошибаются – они делают с точностью то, что заложено программой), то вся цепочка действий (программист, оператор перфокарты, оператор вычислительной машины, проверка результатов) повторялась заново.

Следующий этап в программировании – это появление языка Ассемблера. Этот язык программирования позволял писать довольно длинные для того времени программы. Но Ассемблер – это язык программирования низкого уровня, все операции проводятся на уровне "железа". Если вы не знаете, то сейчас я вам поясню. Чтобы в PHP выполнить простейшее действие, например, сложение, достаточно записать '$A = 2 + 2; '. На языке Ассемблера вам для выполнения этого же действия нужно было выполнить как минимум три действия – загрузить в один из регистров первое число (команда MOV), загрузить в другой регистр второе число (опять команда MOV), выполнить сложение регистров командой ADD. Результат сложения будет помещен в третий регистр. Названия регистров я специально не указывал, поскольку они зависят от архитектуры процессора, а это еще один недостаток Ассемблера. Если вам нужно перенести программу на компьютер с другой архитектурой, вам нужно переписать программу с учетом особенностей целевой архитектуры.

Требования к программным продуктам и к срокам их разработки росли (чем быстрее будет написана программа, тем лучше), поэтому появились языки программирования высокого уровня. Язык высокого уровня позволяет писать программы, не задумываясь об архитектуре вашего процессора. Нет, это не означает, что на любом языке высокого уровня можно написать программу, которая в итоге станет работать на процессоре с любой архитектурой. Просто при написании программы знать архитектуру процессора совсем не обязательно. Вы пишете просто A = B + C и не задумываетесь, в каком из регистров (или в какой ячейке оперативной памяти) сейчас хранятся значения, присвоенные переменным **B** и **C**. Вы также не задумываетесь, куда будет помещено значение переменной **A**. Вы просто знаете, что к нему можно обратиться по имени **A**. Первым языком высокого уровня стал FORTRAN (FORmula TRANslator).

Следующий шаг – это появление структурного программирования. Дело в том, что программы на языке высокого уровня очень быстро стали расти в

размерах, что сделало их нечитабельными из-за отсутствия какой-нибудь четкой структуры самой программы. Структурное программирование подразумевает наличие структуры программы и программных блоков, а также отказ от инструкций безусловного перехода (GOTO, JMP).

После выделения структуры программы появилась необходимость в создании подпрограмм, которые существенно сокращали код программы. Намного проще один раз написать код вычисления какой-то формулы и оформить его в виде процедуры (функции) – затем для вычисления 10 результатов по этой формуле нужно будет 10 раз вызвать процедуру, а не повторять 10 раз один и тот же код. Новый класс программирования стал называться процедурным.

Со временем процедурное программирование постигла та же участь, что и структурное программирование – программы стали настолько большими, что их было неудобно читать. Нужен был новый подход к программированию. Таким стало объектно-ориентированное программирование (далее ООП).

ООП базируется на трех основных принципах – инкапсуляция, полиморфизм, наследование. Разберемся, что есть что.

С помощью инкапсуляции вы можете объединить воедино данные и обрабатывающий их код. Инкапсуляция защищает и код, и данные от вмешательства извне. Базовым понятием в ООП является класс. Грубо говоря, класс – это своеобразный тип переменной. Экземпляр класса (переменная типа класс) называется объектом. В свою очередь, объект – это совокупность данных (свойств) и функций (методов) для их обработки. Данные и методы обработки называются членами класса. Свойства в Python называются атрибутами класса, но также и есть понятие свойства класса, которое не нужно путать с классическим свойством.

Получается, что объект – это результат инкапсуляции, поскольку он включает в себя и данные, и код их обработки. Чуть дальше вы поймете, как это работает, пока представьте, что объект – это эдакий рюкзак, собранный по принципу "все свое ношу с собой".

Теперь поговорим о полиморфизме. Если вы программировали на языке C (на обычном C, не C++), то наверняка знакомы с функциями *abs()*, *fabs()*, *labs()*. Все они вычисляют абсолютное значение числа, но каждая из функций используется для своего типа данных. Если бы C поддерживал полиморфизм, то можно было бы создать одну функцию *abs()*, но объявить ее трижды – для каждого типа данных, а компилятор бы уже сам выбирал нужный вариант функции, в зависимости от переданного ей типа данных. Данная практика называется перезагрузкой функций. Перезагрузка

функций существенно облегчает труд программиста – вам нужно помнить в несколько раз меньше названий функций для написания программы.

Полиморфизм позволяет нам манипулировать с объектами путем создания стандартного интерфейса для схожих действий.

Осталось поговорить о наследовании. С помощью наследования один объект может приобретать атрибуты другого объекта. Заметьте, наследование – это не копирование объекта. При копировании создается точная копия объекта, а при наследовании эта копия дополняется уникальными атрибутами (новыми членами). Наследование можно сравнить с рождением ребенка, когда новый человек наследует атрибуты своих родителей, но в то же время не является точной копией одного из родителей.

16.2. Определение класса и создание объекта

Определить класс можно с помощью ключевого слова **class**:

```
class <Название класса>:
  <Описание атрибутов и методов>
```

Пример определения класса:

```
class SampleClass:
  def __init__(self):
    print("Constructor")
    self.nm = "SampleClass"
  def printName(self):
    print(self.nm)

obj = SampleClass()

obj.printName()

# При желании вы можете самостоятельно вывести атрибут
print(obj.nm)
```

Если запустить этот код, то его вывод будет следующим:

```
>>>
Constructor
SampleClass
SampleClass
```

Строка **Constuctor** выводит только один раз – во время создания объекта **obj**. Далее выводятся две строки SampleClass. Одна – когда мы используем метод *printName()*, вторая – когда мы выводим атрибут объекта.

Как видите, формат обращения к методам и атрибутам следующий:

```
<Объект>.<Метод>([Параметры])
<Объект>.<Атрибут>
```

Также для доступа к атрибутам вы можете использовать следующие функции:

- *getattr()* — возвращает значение атрибута по его названию, которое указывается в виде строки
- *setattr()* — устанавливает значение атрибута. Название атрибута задается в виде строки
- *delattr()* — удаляет атрибут, название, как обычно, задается в виде строки
- *hasattr()* — проверяет наличие указанного атрибута. Если атрибут существует, возвращается *True*

Синтаксис данных функций следующий:

```
getattr(<Объект>, <Атрибут>[, <Значение по умолчанию>])
setattr(<Объект>, <Атрибут>, <Значение>)
delattr(<Объект>, <Атрибут>)
hasattr(<Объект>, <Атрибут>)
```

16.3. Конструктор и деструктор

Конструктор – это метод, вызываемый интерпретатором автоматически при инициализации класса. В Python этот метод называется __init__():

```
def __init__(self[, <Значение1>[, …, <ЗначениеN>]]:
   <Инструкции>
```

Конструктор используется для инициализации атрибутов класса. Также конструкторы могут выполнять некоторую подготовительную работу, например, открывать файлы, устанавливать соединения – все зависит от специфики вашей программы.

Аналогично, перед уничтожением объекта вызывается деструктор, который в Python называется `__del__()`. Учитывая, что интерпретатор сам заботится об освобождении занимаемых объектом ресурсов, особого смысла в деструкторе нет.

16.4. Наследование

Наследование – это то, за что программисты любят ООП. Наследование позволяет создать класс, в котором будет доступ ко всем атрибутам и методам родительского (базового) класса, а также к некоторым новым методам и атрибутам.

Рассмотрим небольшой пример:

```
class Parent:                # Родительский класс
  def print_name(self):
    print("Родитель")

class Child(Parent):         # Наследование класса Parent
  def print_child(self):
    print("Потомок")

obj = Child()
obj.print_name()
obj.print_child()
```

Посмотрите, что у нас получилось. Класс Child унаследовал метод *print_name()*, который мы можем вызвать из объекта **obj**.

> ***Примечание.*** Терминология разная. Поэтому в некоторых источниках родительский класс могут называть базовым, а также суперклассом. Дочерний класс называют подклассом или производным классом.

А что, если в дочернем классе вам захочется определить такой же метод, как и в родительском? Например:

```
class Parent:                # Родительский класс
  def print_name(self):
    print("Родитель")

class Child(Parent):         # Наследование класса Parent
```

```
    def print_child(self):
        print("Потомок")
    def print_name(self):
        print("Потомок")

obj = Child()
obj.print_name()
```

Какой метод будет вызван? Будет вызван метод дочернего класса, поскольку он переопределит метод с таким же именем родительского класса. Если нужно вызвать именно метод родительского класса, тогда нужно явно указать имя класса, например:

```
    def print_name(self):
        print("Потомок")
        Parent.print_name()
```

Примечание. Конструктор родительского класса автоматически не вызывается, если он переопределен в дочернем классе!

В Python также доступно и множественное наследование – когда один класс наследует атрибуты и методы нескольких классов. Просто нужно указать родительские классы в скобках через запятую:

```
class Child(Parent1, Parent2):
    <Определение класса, как обычно>
```

16.5. Специальные методы

Классы в Python поддерживают представленные в таблице 16.1 специальные методы.

Таблица 16.1. Специальные методы

Метод	Описание
__call__(self[, Параметр1,... ,ПараметрN])	Обрабатывает вызов экземпляра класса как вызов функции

__setitem__(self, <Ключ>, <Значение>)	Будет вызван при присваивании значения по индексу или ключу
__getitem__(self, <Ключ>)	Вызывается при доступе к значению по индексу или ключу. Метод будет автоматически вызван при использовании операций, применимых к последовательностям, например, при использовании цикла **for**
__delitem__(self, <Ключ>)	Вызывается при удалении элемента по индексу или ключу с помощью оператора **del**
__getattr__(self, <Атрибут>)	Вызывается при обращении к несуществующему атрибуту класса
__getattribute__(self, <Атрибут>)	Вызывается при обращении к любому атрибуту класса
__setattr__(self, <Атрибут>, <Значение>)	Вызывается при попытке присваивания значения атрибуту экземпляра класса
__delattr__(self, <Атрибут>)	Вызывается при удалении атрибута с помощью инструкции de1 <Экземпляр класса>.<Атрибут>
__iter__(self)	Определяется только для объектов, поддерживающих итерацию. Если в классе одновременно определены методы __iter__() и __getitem__(), то предпочтение отдается методу __iter__(). Помимо метода __iter__() в классе должен быть определен метод __next__(), который будет вызываться на каждой итерации
__len__(self)	Вызывается при использовании функции len()

__bool__(self)	Вызывается при использовании функции bool()
__int__(self)	Используется при преобразовании объекта в целое число с помощью функции int()
__float__(self)	Используется при преобразовании объекта в целое число с помощью функции float()
__complex__(self)	Вызывается при использовании функции complex()
__round__(self)	Вызывается при использовании функции round()
__index__(self)	Вызывается при использовании функций bin(), hex() и oct()
__repr__(self)	Используется для преобразования объекта в строку. Вызывается при попытке преобразовать объект в строку в интерактивной оболочке, а также при использовании функции repr()
__str__(self)	Используется для преобразования объекта в строку. Вызывается при попытке преобразовать объект в строку при выводе объекта функцией print(), а также при использовании функции str(). Если метод __str__() не определен, то будет вызван метод __repr__(). Методы __str__() и __repr__() должны возвращать строку
__hash__(self)	Используется, если объект класса планируется использовать в качестве ключа словаря или внутри множества. Используется редко

Вы можете переопределить эти специальные методы так, как посчитаете нужным – в зависимости от решаемой задачи. В качестве примера создадим класс, поддерживающий итерацию. Для этого нам нужно переопределить методы __iter__() и __next__():

```
class IterClass:
    def __init__(self, x):
        self.massiv = x
        self.ind = 0    # Индекс
    def __iter__(self):
        return self
    def __next__(self):
        if self.ind >= len(self.massiv):
            self.ind = 0              # Сбрасываем индекс
            raise StopIteration       # Генерируем исключение
        else:
            item = self.massiv[self.ind]
            self.ind += 1
            return item

obj = IterClass([1, 2, 3])
for i in obj:
    print(i, end=" ") # выведет 1 2 3
```

16.6. Статические методы

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса. Для определения статического метода используется декоратор @staticmethod. Вызывается статический метод так:

```
<Название класса>.<Название метода>(<Параметры>)
```

Также статический метод можно вызвать и через объект класса:

```
<Объект класса>.<Название метода>(<Параметры>)
```

Пример:

```
class StaticSample:
    @staticmethod
    def ssum(x, y):
        return x + y
    def msum(self, x, y):
        return x + y
```

```
print(StaticSamle.ssum(2, 2))    # Вызываем до объявления
объекта
obj = StaticSample()
print(obj.msum(2, 2))            # Вызываем обычный метод
print(obj.ssum(2, 2))            # Вызываем статический метод
через объект
```

16.7. Абстрактные методы

Абстрактные методы содержат только определение метода без реализации. Это эдакие заглушки, которые нужно реализовать в дочернем классе. Часто в абстрактных методах возбуждают исключение, чтобы напомнить о необходимости реализовать метод, например:

```
class Sample:
  def func(self, x, y):
    raise NotImplementedError("Not implemented")
def msum(self, x, y):
    return x + y
```

В данном случае метод *func()* является абстрактным. Как видите, никаких декораторов не используется. Хотя в версии 2.6 появился модуль **abc**, содержащий декоратор @abstractmethod. Что дает нам использование этого декоратора? А то, что вам не нужно вызывать самому исключение, данный декоратор сгенерирует ошибку TypeError при использовании не переопределенного абстрактного метода. Лучше использовать первый способ, но не привести пример с использованием @abstractmethod просто невозможно:

```
from abc import *

class Sample:
  @abstractmethod
  def func(self, x, y):
    pass
def msum(self, x, y):
    return x + y
```

16.8. Перегрузка операторов

Перегрузка обычных операторов позволяет экземплярам классов участвовать в обычных операциях вроде сложения или вычитания. Например,

если вы хотите сложить два объекта, то для их класса должен быть определен метод x.__add__(y). Ниже приведен пример перегрузки операторов == и in:

```
class ReloadClass:
   def __init__(self):
      self.x = 0
      self.a = [1, 2, 3]
   def __eq__(self, x):
      return self.x == y
   def __contains__(self, y):
      return y in self.a

o = ReloadClass()
if o == 10:
   print("True")
else:
   print("False")          # Выведет False

if 3 in o:
   print("True")           # Выведет True
else:
   print("False")
```

Методы, используемые для перегрузки обычных операторов, приведены в таблице 16.2.

Таблица 16.2. Методы перезагрузки обычных операторов

Метод	**Оператор**
x.__add__(y)	x + y
x.__radd__(y)	y + x (экземпляр класса справа)
x.__iadd__(y)	x += y
x.__sub__(y)	x - y
x.__rsub__(y)	y - x (экземпляр класса справа)
x.__isub__(y)	x -= y
x.__mul__(y)	x * y
x.__rmul__(y)	y * x (экземпляр класса справа)

x.__imul__(y)	x *= y
x.__truediv__(y)	x / y
x.__rtruediv__(y)	y / x
x.__itruediv__(y)	x /= y
x.__floordiv__(y)	x // y
x.__rfloordiv__(y)	y // x
x.__iflordiv__(y)	x //= y
x.__mod__(y)	x % y
x.__rmod__(y)	y % x
x.__imod__(y)	x %= y
x.__pow__(y)	x ** y
x.__rpow__(y)	y ** x
x.__ipow__(y)	x **= y
x.__neg__()	-x (унарный минус)
x.__pos__()	+x (унарный плюс)
x.__abs__()	abs(x)
x.__contains__(y)	in
x.__eq__(y)	x == y
x.__ne__(y)	x != y
x.__lt__(y)	x < y
x.__gt__(y)	x > y
x.__le__(y)	x <= y
x.__ge__(y)	x >= y

16.9. Свойства класса

Внутри класса может быть создан идентификатор, через который будут производиться операции по получению и изменению значения атрибута, а также операция удаления атрибута. Создать такой идентификатор моно с помощью функции *property()*:

```
<Свойство> = property(<Чтение>[, <Запись>[, <Удаление[, <Строка>]]])
```

Первые три параметра определяют соответствующий метод класса. При чтении значения будет вызван метод, указанный в первом параметре. При попытке записи – метод, указанный во втором параметре. При удалении атрибута вызывается метод, указанный в третьем параметре. Если в качестве какого-либо параметра указано *None*, то это означает, что этот метод не поддерживается. Последний параметр – это строка документирования.

Пример:

```
class PropertySampleClass:
  def __init__(self, x):
    self.__p = x
  def get_p(self):
    return self.__p
  def set_p(self, x):
    self.__p = x
  def del_p(self):
    del self._p
  prop = property(get_p, set_p, del_p, "Info")

o = PropertySampleClass(1)
print(o.prop)              # Вызывается метод get_p
o.prop = 5                 # Вызывается метод set_p
del o.prop                 # Вызывается метод del_p
```

16.10. Декораторы класса

Начиная с Python 3, кроме декораторов функций поддерживаются также декораторы классов, позволяющие изменить поведение обычных классов. В качестве параметра декоратор принимает ссылку на объект класса, поведение которого необходимо изменить, и должен возвращать ссылку на тот же класс или какой-либо другой. Пример:

```
def deco(d):
```

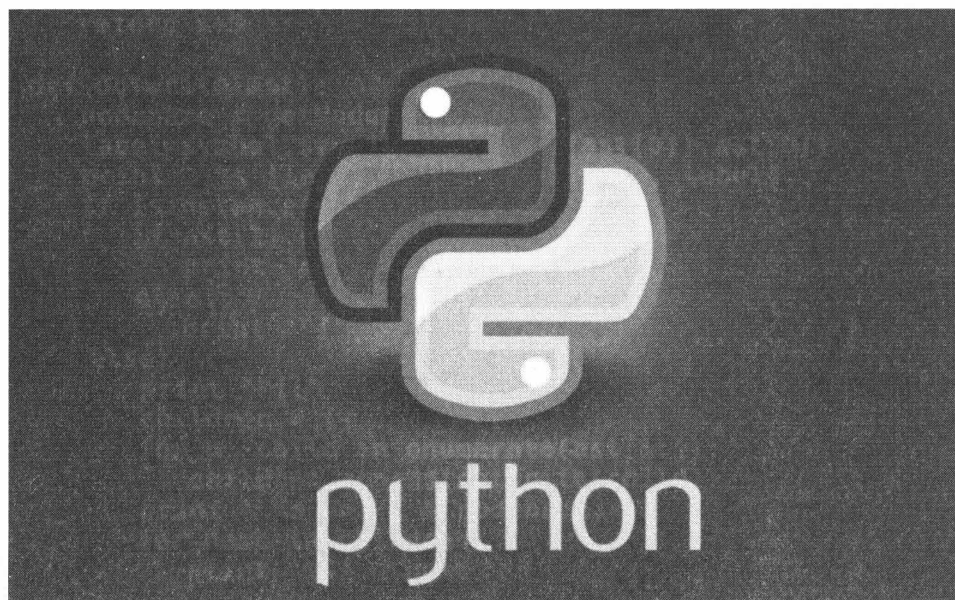
```
    print("Декоратор")
    return d

@deco
class SampleClass:
    def __init__(self, value):
        self.v = value

o = SampleClass(1)
print(o.v)
```

ГЛАВА 17.

РАБОТА С ИНТЕРНЕТОМ



17.1. Разбираем URL-адреса

При написании Интернет-приложений часто может понадобиться разбор URL-адреса. Общий формат URL следующий:

```
<Протокол>://<Домен>:<Порт>/<Путь>;<Параметры>?<Запрос>#<Яко
рь>
```

Для протокола FTP схема URL выглядит так:

```
<Протокол>://<Пользователь>:<Пароль>@<Домен>
```

Для разбора URL в Python используется функция *urlparse()* из модуля **urlib**.parse. Синтаксис ее следующий:

```
urlparse(<URL-адрес>[, <Схема<[, <Якорь>]])
```

Функция возвращает объект ParseResult с результатами разбора URL-адреса. Данный объект можно преобразовать в кортеж из следующих элементов (scheme, netloc, path, params, query, fragment). Данные элементы соответствуют схеме URL-адреса:

```
<scheme>://<netloc>/<path>;<params>?<query>#<fragment>
```

Элементы:

- **scheme** – содержит название протокола. По умолчанию пустая строка. Доступно по индексу 0
- **netloc** – название домена и номер порта. По умолчанию пустая строка. Доступно по индексу 1
- **hostname** – только название домена (в нижнем регистре)
- **port** – номер порта
- **path** – путь. Доступно по индексу 2
- **params** – параметры. Значение доступно по индексу 3. По умолчанию – пустая строка
- **query** – строка запроса. Значение доступно по индексу 4
- **fragment** – якорь. По умолчанию – пустая строка. Значение доступно по индексу 5
- **username** – имя пользователя (если указано), по умолчанию – *None*
- **password** – пароль. Значение по умолчанию – *None*

Пример использования:

```
>>> from urllib.parse import *
>>> url = urlparse("http://nit.center:80/index.php;st?param=value#ankor")
>>> url
ParseResult(scheme='http', netloc='nit.center:80', path='/index.php',
params='st', query='param=value', fragment='ankor')
>>> t = tuple(url)
>>> t
('http', 'nit.center:80', '/index.php', 'st', 'param=value', 'ankor')
>>> url.scheme, url[0]
('http', 'http')
>>> url.netloc, url[1]
```

```
('www.nit.center:80', 'www.nit.center:80')
>>> url.hostname
'nit.center'
>>> url.port
80
>>> url.path
'/index.php'
>>> url.params
'st'
>>> url.query
'param=value'
>>> url.fragment
'ankor'
>>>
```

Для выполнения обратной операции, то есть для сбора URL-адреса из отдельных частей используется функция *urlunsplit()*:

```
>>> t = ('http', 'nit.center', 'index.php', 'par=value', 'ankor')
>>> urlunsplit(t)
'http://nit.center/index.php?par=value#ankor'
```

17.2. Декодирование строки запроса

Ранее мы получили строку запросов, которая выглядит так:

```
параметр1=значение1&...&параметрN=значениеN
```

Теперь нужно разобрать эту строку на составляющие. Сложность заключается в том, что если значение содержит символы национальных алфавитов, то они будут закодированы – каждый символ будет кодироваться последовательностью символов $nn, где **nn** – шестнадцатеричное число. Пример:

https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5

Для разбора строки запроса на составляющие можно использовать следующие функции из модуля urllib.parse:

- **parse_qs()** – разбирает строку запроса и возвращает словарь с ключами, которые содержат названия параметров, и список значений
- **parse_qsl()** – в отличие от предыдущей функции возвращает список кортежей из двух элементов

Синтаксис функций:

```
urlparse.parse_qs(qs[, keep_blank_values[, strict_parsing]])
urlparse.parse_qsl(qs[, keep_blank_values[, strict_parsing]])
```

Пример:

```
>>> from urllib.parse import parse_qs
>>> qs = 's=%D0%A8%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D
0%B8%D0%B5'
>>> parse_qs(qs, encoding='cp1251')
{'s': ['PЁPёC„CЂPsPIP°PSPёPµ']}
>>> parse_qs(qs, encoding='utf-8')
{'s': ['Шифрование']}
>>>
```

Как видно из примера, очень важно правильно указать кодировку. Как правило, используется **utf-8**, что и показано в примере.

Если параметров несколько, то функции с легкостью справятся и с этой задачей:

```
>>> qs = 'par1=val1&par2=val2'
>>> parse_qs(qs, encoding='utf-8')
{'par1': ['val1'], 'par2': ['val2']}
>>> parse_qsl(qs, encoding='utf-8')
[('par1', 'val1'), ('par2', 'val2')]
>>>
```

17.3. Разбор HTML-эквивалентов

В HTML-документах часто встречаются так называемые HTML-эквиваленты. Например, последовательность &gt соответствует знаку >, а

последовательность &lt – знаку <. Для работы с HTML-эквивалентами используются функции модуля xml.sax.saxutils:

- **escape()** – заменяет символы <, >, & соответствующим им HTML-эквивалентами. Необязательный параметр <Словарь> позволяет указать словарь, содержащий дополнительные символы в качестве ключей, а их HTML-эквиваленты – в качестве значений.
- **quoteattr()** – аналогична escape(), но дополнительно заключает строку в кавычки или апострофы. Если внутри строки встречаются и кавычки, и апострофы, то двойные кавычки будут заменены HTML-эквивалентом.
- **unescape()** – заменяет HTML-эквиваленты &, <, > обычными символами. Параметр <Словарь> позволяет указать словарь с дополнительными эквивалентами.

Синтаксис:

```
escape(<Строка>[, <Словарь>])
quoteattr(<Строка>[, <Словарь>])
unescape(<Строка>[, <Словарь>])
```

Пример:

```
>>> import xml.sax.saxutils as x
>>> x.escape('""<>&""')
'""&lt;&gt;&amp;""'
```

17.4. Преобразование относительных ссылок

Иногда в HTML-документах указывают относительные ссылки, а не абсолютные. Преобразовать относительную ссылку в абсолютную можно с помощью функции *urljoin()* из модуля urllib.parse:

```
urljoin(<базовый url>, <относительный или абсолютный url> [,
<Якорь>])
```

Пример использования:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://nit.center', 'index.php')
'http://nit.center/index.php'
>>> urljoin('http://nit.center/index.php', 'test.html')
'http://nit.center/test.html'
```

17.5. Определение кодировки

Документы в Интернете представлены в самых разных кодировках. В последнее время наблюдается тенденция использования единой кодировки – UTF-8 и это очень и очень хорошо. Определить кодировку документа можно по заголовку Content-Type в заголовках сервера:

```
Content-Type: text/html; charset=utf-8
```

Также кодировка указывается с помощью META-тега content:

```
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
```

К сожалению, далеко не все используют UTF-8 и довольно часто кодировка из ответа сервера не совпадает с указанной в META-теге. Для определения кодировки можно использовать библиотеку chardet, скачать которую можно по адресу *http://chardet.feedparser.org/download*.

Скачайте архив python3-chardet-2.0.1.tgz и распакуйте его в любую папку. Представим, что вы ее распаковали в папку C:\Python39. Далее откройте командую строку и введите команду:

```
cd C:\Python39\python3-chardet-2.0.1
```

После этого нужно запустить программу установки:

```
cd c:\Python39\python.exe setup.py install
```

После установки библиотеки ее можно использовать:

```
>>> import chardet
```

Для определения кодировки используется функция detect(<последовательность байтов>). В качестве результата функция возвращает словарь с элементами – *encoding* и *confidence*. Первый – это определенная кодировка, второй – вещественное число, задающее коэффициент точности определения кодировки.

Пример:

```
>>> import chardet
>>> chardet.detect(bytes("Текст", "cp1251"))
{'confidence':0.99, 'encoding': 'windows-1251'}
```

Как видите, мы передали функции строку байтов в кодировке Windows-1251, и она правильно ее определила с точностью 99%.

17.6. Реализация HTTP-клиента

Представим, что есть сценарий *http://nit.center/script.php*. Ему нужно передать параметры var1 и var2, а потом считать результат его работы.

В простых случаях вполне достаточно будет модуля urllib.request. Например, для отправки простого запроса HTTP GET удаленному сервису достаточно сделать следующее:

```
from urllib import request, parse
# URL, к которому производится доступ
url = 'http://nit.center/script.php'
# Словарь с параметрами запроса (если есть)
parms = {
  'var1' : 'value1',
  'var2' : 'value2'
}
# Кодируем строку запроса
querystring = parse.urlencode(parms)
```

```
# Производим GET-запрос и читаем ответ
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

Если вам нужно отправить параметры запроса в теле POST-запроса, сначала закодируйте их и отправьте в виде опционального аргумента urlopen():

```
from urllib import request, parse
# URL, к которому производится доступ
url = 'http://nit.center/script.php'
# Словарь с параметрами запроса (если есть)
parms = {
  'name1' : 'value1',
  'name2' : 'value2'
}
# Кодируем строку запроса
querystring = parse.urlencode(parms)

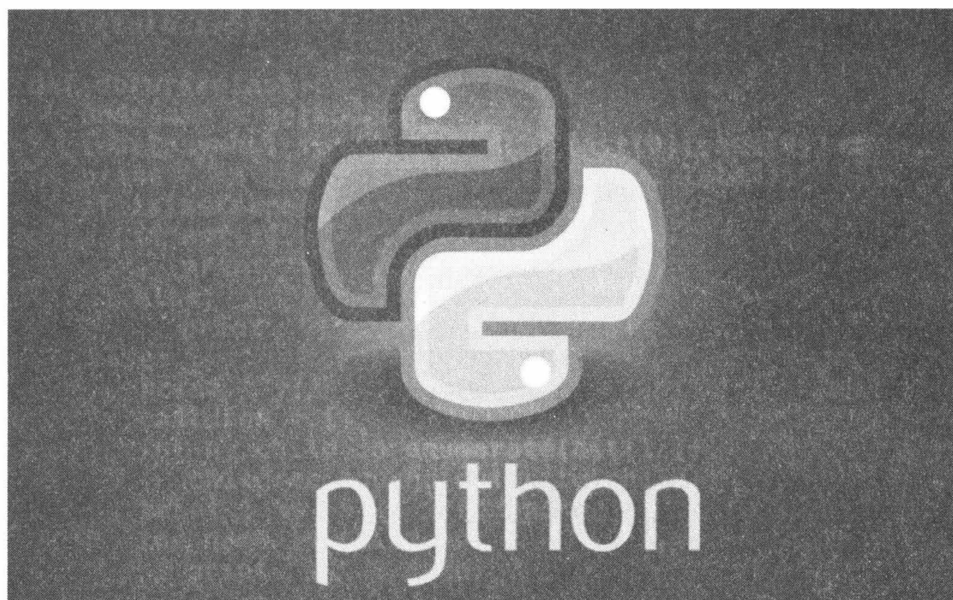
# Производим POST-запрос и читаем ответ
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

Если вам нужно предоставить пользовательские HTTP-заголовки в исходящем запросе, например, изменить поле User-Agent (обычно используется для указания браузера), создайте словарь, содержащий их значения, создайте экземпляр **Request** и передайте его в *urlopen()*. Например:

```
from urllib import request, parse
...
# Дополнительные заголовки
headers = {
  'User-agent' : 'My Python Browser'
}
req = request.Request(url, querystring.encode('ascii'),
headers=headers)
# Производим запрос и читаем ответ
u = request.urlopen(req)
resp = u.read()
```

ГЛАВА 18.

ИТЕРАТОРЫ И ГЕНЕРАТОРЫ



Итерация – одна из самых сильных функций Python. На высоком уровне вы можете просто рассматривать итерацию как способ обработки элементов в последовательности. Однако вам доступно намного больше, например, создание собственных объектов **iterator** (итераторов), применение полезных итеративных образцов в модуле **itertools**, создание функций-генераторов и т.д. Эта глава стремится лишь показать типичные задачи, вовлекающие итерацию.

18.1. Ручное использование итератора

Начнем с самого простого примера: вам нужно обработать элементы в итерируемом, но по некоторым причинам вы не можете или не хотите использовать цикл **for**.

Используйте функцию *next()* и напишите код для перехвата исключения *StopIteration*. Например, этот пример читает все строки из файла:

```
with open('cron.log') as f:
  try:
     while True:
          line = next(f)
          print(line, end='')
  except StopIteration:
       pass
```

Обычно исключение *StopIteration* используется для уведомления об окончании итерации. Однако, если вы будете использовать *next()* вручную (как и показано), вы можете также возвращать какое-то завершающееся значение вроде *None*. Например:

```
with open('cron.log') as f:
    while True:
        line = next(f, None)
        if line is None:
            break
        print(line, end='')
```

В большинстве случаев для перебора итерируемого используется оператор **for**. Однако время от времени задачи будут требовать более точного управления итеративным механизмом. Таким образом, полезно знать, что фактически происходит.

Следующий интерактивный пример иллюстрирует основную механику того, что происходит во время итерации:

```
>>> items = [1, 2, 3]
>>> # Получаем итератор
>>> it = iter(items) # Вызываем items.__iter__()
>>> # Запускаем итератор
>>> next(it) # Вызываем it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Последующие примеры в этой главе подробно останавливаются на итеративных методах и подразумевают, что вы знакомы с основным протоколом итератора.

18.2. Делегирование итерации

Представим, что у нас есть пользовательский объект контейнера, который внутри содержит список, кортеж или какое-то другое итерируемое. Нужно проделать итерацию с вашим новым контейнером.

Как правило, все, что вам нужно сделать – это определить метод __iter()__, который делегирует итерацию к внутреннему контейнеру. Например:

```
class Node:
     def __init__(self, value):
         self._value = value
         self._children = []

     def __repr__(self):
         return 'Node({!r})'.format(self._value)

     def add_child(self, node):
         self._children.append(node)

     def __iter__(self):
         return iter(self._children)

# Пример
if __name__ == '__main__':
   root = Node(0)
   child1 = Node(1)
   child2 = Node(2)
   root.add_child(child1)
   root.add_child(child2)
   for ch in root:
       print(ch)
# Выводит Node(1), Node(2)
```

В этом коде метод __iter__() просто перенаправляет запрос итерации к внутреннему атрибуту _children.

Протокол итератора в Python требует __iter__() для возврата специального объекта итератора, который реализует метод __next__() для выполнения фактической итерации.

Если все, что вы делаете, это просто итерация по содержимому другого контейнера, вам не нужно волноваться о том, как это работает. Все, что вы должны сделать, это перенаправить запрос итерации вперед.

Использование функции iter() здесь что-то вроде ярлыка, который делает код чище. Функция просто возвращает базовый итератор, вызывая метод s.__iter__().

18.3. Создание нового шаблона итерации с помощью генераторов

В этом примере мы попытаемся реализовать пользовательский шаблон итерации, который отличается от обычных встроенных функций (например, *range()*, *reversed()* и т.д.).

Если вам нужно реализовать новый вид шаблона итерации, определите его как функцию-генератор. Здесь приведен генератор, который создает диапазон чисел с плавающей точкой:

```
def my_range(start, stop, increment):
    x = start
    while x < stop:
       yield x
       x += increment
```

Чтобы использовать эту функцию, вам нужно итерировать по ней в цикле

for или использовать ее с другой функцией, работающей с итерируемым (например, *sum()*, *list()* и т.д.). Например:

```
>>> for n in my_range(0, 4, 0.5):
... print(n)
...
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(my_range(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

Простое присутствие оператора **yield** в функции превращает ее в генератор. В отличие от обычной функции, генератор работает только в ответ на итерацию. Ради эксперимента рассмотрим, как работает такая функция:

```
>>> def countdown(n):
... print('Начальная позиция отсчета: ', n)
... while n > 0:
...     yield n
...     n -= 1
... print('Готово!')
...
>>> # Создаем генератор, обратите, что не будет никакого
вывода
>>> c = countdown(3)
>>> c
<generator object countdown at 0x1006a0af0>
>>> # Запускаем генератор до первого yield и получаем значение
>>> next(c)
Начальная позиция отсчета 3
3
>>> # Запуск до следующего yield
>>> next(c)
2
>>> # Запуск до следующего yield
>>> next(c)
1
>>> # # Запуск до следующего yield (итерация останавливается)
>>> next(c)
Готово!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Главная особенность – то, что функция генератора только работает в ответ на "следующие" операции, выполненные в итерации. Как только функция возвращается, итерация останавливается. Однако обычно цикл **for** заботиться обо всех деталях, поэтому вас это не должно волновать.

18.4. Реализация протокола итератора

При создании пользовательских объектов, поддерживающих итерацию, полезно реализовать и протокол итератора.

Безусловно, самый простой способ реализовать итерацию на объекте заключается в использовании функции-генератора.

Ранее был разработан класс Node для представления структур деревьев. Возможно, вы хотите реализовать итератор, который делает обход узлов в глубину (метод будет называться *depth_first*). Вот как это можно сделать

```
class Node:
     def __init__(self, value):
         self._value = value
         self._children = []

     def __repr__(self):
         return 'Node({!r})'.format(self._value)

     def add_child(self, node):
         self._children.append(node)

     def __iter__(self):
         return iter(self._children)

    def depth_first(self):
         yield self
         for c in self:
             yield from c.depth_first()

# Пример
if __name__ == '__main__':
    root = Node(0)
    childl = Node(1)
    child2 = Node(2)
    root.add_child(childl)
    root.add_child(child2)
    childl.add_child(Node(3))
    childl.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
```

```
    # Выведет Node(0), Node(1), Node(3), Node(4), Node(2),
Node(5)
```

Метод *depth_first()* прост. Сначала он выполняет оператор *yield self*, а затем итерирует по каждому дочернему элементу, отправляя с помощью **yield** каждый дочерний элемент, произведенный методом *depth_first()* дочернего элемента (используя *yield from*).

Протокол итератора Python требует метод __iter__() для возврата специального объекта итератора, который реализует операцию __next__() и использует исключение *StopIteration* для уведомления о завершении итерации. Однако реализация таких объектов может часто быть "грязным" делом. Например, следующий код показывает альтернативную реализацию метода *depth_first()*, используя связанный класс **iterator**:

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, other_node):
        self._children.append(other_node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)

class DepthFirstIterator(object):

'''
Обход в глубину
'''
    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
```

```
        self._child_iter = None

    def __iter__(self):
        return self

    def __next__(self):
        # Возвращает себя, если только запущен. Создает итератор для дочерних
        # объектов
        if self._children_iter is None:
                self._children_iter = iter(self._node)
                return self._node

        # Если обрабатывается дочерний объект, возвращает его следующий элемент
        elif self._child_iter:
                try:
                        nextchild = next(self._child_iter)
                        return nextchild
                except StopIteration:
                        self._child_iter = None
                        return next(self)
        # Переход к следующему потомку и запуск его итерации
        else:
                self._child_iter = next(self._children_iter).depth_first()
                return next(self)
```

Класс *DepthFirstIterator* работает так же, как и версия с генератором, но это – путаница, потому что итератор должен обслуживать больше сложного состояния о том, где он находится в процессе итерации. Откровенно говоря, никому не нравится писать такой взрывающий мозг код, как этот. Определите свой итератор, как генератор и жизнь станет проще.

18.5. Итерация в обратном направлении

Для итерации в обратном направлении используйте встроенную функцию *reversed()*. Например:

```
>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...
4
3
2
1
```

Обратная итерация работает только, если объект имеет размер, который может быть определен или же у объекта реализован специальный метод __reversed__(). Если для вашего объекта не выполняется ни то, ни другое, вам нужно сначала конвертировать ваш объект в список. Например:

```
# Выводим файл в обратном порядке
f = open('file.txt')
for line in reversed(list(f)):
    print(line, end='')
```

Знайте, что превращение итерируемого в список, как показано выше, может требовать огромного количества памяти, если итерируемое большое.

Многие программисты не понимают, что обратная итерация может быть настроена с помощью определяемых пользователем классов, если они реализуют метод __reversed__(). Например:

```
class Countdown:
      def __init__(self, start):
            self.start = start

      # Итератор вперед
      def __iter__(self):
            n = self.start
            while n > 0:
                  yield n
                  n -= 1
      # Итератор назад
      def __reversed__(self):
            n = 1
            while n <= self.start:
                  yield n
                  n += 1
```

Определение обратного итератора делает код намного более эффективным, поскольку больше нет необходимости преобразовывать данные в список и итерировать по списку в обратном направлении.

18.6. Экстра-состояние функции-генератора

Если вы хотите, чтобы генератор представил дополнительное состояние пользователю, не забывайте, что вы можете легко реализовать его, как класс, поместим код функции-генератора в метод __iter__(). Например:

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines,1):
            self.history.append((lineno, line))
        yield line

    def clear(self):

        self.history.clear()
```

Чтобы использовать этот класс, смотрите на него как на обычную функцию-генератор. Однако, поскольку она создает экземпляр, вы можете получить доступ к внутренним атрибутам, таким как атрибут **history** или метод *clear()*. Например:

```
with open('somefile.txt') as f:
    lines = linehistory(f)
    for line in lines:
        if 'python' in line:
            for lineno, hline in lines.history:
                print('{}:{}'.format(lineno, hline),
            end='')
```

Используя генераторы, легко попасть в ловушку, попытавшись сделать все только функциями. Это может привести к очень сложному коду, особенно если функция-генератор должна взаимодействовать с другими частями вашей программы необычными способами (предоставлять атрибуты, разрешать управление через вызов метода и т.д.). Если это так, просто используйте определение класса, как показано выше. При определении генератора в методе __iter__() не нужно изменять свой алгоритм. Тот факт, что это часть класса упрощает доступ пользователей к атрибутам и методам, чтобы взаимодействовать с ними.

Есть одна потенциальная тонкость с показанным методом – то, что он мог бы потребовать дополнительный шаг, заключающийся в вызове *iter()*, если вы собираетесь управлять итерация, используя технику, отличную от цикла **for**. Например:

```
>>> f = open('file.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator
>>> # Сначала вызываем iter(), затем запускаем итерацию
>>> it = iter(lines)
>>> next(it)
'line 1\n'
>>> next(it)
'line 2\n'
>>>
```

18.7. Пропуск первой части итерируемого

Модуль **itertools** содержит несколько функций, которые могут использоваться для решения этой задачи. Первой является функция *itertools.dropwhile()*, которой нужно передать функцию и итерируемое. Возвращенный итератор отбрасывает первые элементы в последовательности, пока предоставленная функция не вернет *true*.

Чтобы проиллюстрировать использование этой функции, представим, что у нас есть файл, который начинается серией комментариев. Например:

```
>>> with open('test.txt') as f:
...       for line in f:
...           print(line, end='')
...
# Comment line 1
# Comment line 2
Line 1
Line 2
Line 3
>>>
```

Если вам нужно пропустить все начальные комментарии, вам нужно сделать это:

```
>>> from itertools import dropwhile
>>> with open('test.txt') as f:
...       for line in dropwhile(lambda line: line.
startswith('#'), f):
...           print(line, end='')
...
Line 1
Line 2
Line 3
>>>
```

Этот пример основан на пропуске первых элементов в соответствии с функцией **test**. Если вы знаете точное число элементов, которые вы хотите пропустить, вы можете использовать другую функцию – *itertools.islice()*. Например:

```
>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...      print(x)
...
1
4
10
15
>>>
```

В этом примере последний аргумент *None* к islice() требуется, чтобы указать, что вы хотите получить все после первых трех элементов в противопо-

ложность только первым трем элементам (например, часть [3:] в в противоположность части [:3]).

18.8. Итерирование по всем возможным комбинациям или перестановкам

Модель **itertools** предоставляет три функции для этой задачи. Первая из

них – *itertools.permutations()* принимает коллекцию элементов и производит последовательность кортежей, содержащую все возможные перестановки элементов (то есть она перемешивает коллекцию во все возможные конфигурации). Например:

```
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
...     print(p)
...
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
        print(p)


('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
Ln: 15 Col: 4
```

Рис. 18.1. Генерирование перестановок

Если вы хотите получить все перестановки меньшей длины, вы можете задать необязательный параметр длины. Например:

```
>>> for p in permutations(items, 2):
...     print(p)
...
('a', 'b')
('a', 'c')

('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>
```

Функция *itertools.combinations()* используется для создания последовательности комбинаций элементов, взятых при вводе. Например:

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
... print(c)
...
('a', 'b', 'c')
>>> for c in combinations(items, 2):
... print(c)
...
('a', 'b')
('a', 'c')
('b', 'c')
>>> for c in combinations(items, 1):
... print(c)
...
('a',)
('b',)
('c',)
>>>
```

Для *combinations()* не рассматривается актуальный порядок элементов. Поэтому комбинация ('a', 'b') рассматривается как аналогичная ('b', 'a') и не выводится.

При создании комбинаций выбранные элементы удаляются из коллекции возможных кандидатов (то есть, если 'a' уже выбрана, то она больше не рассматривается).

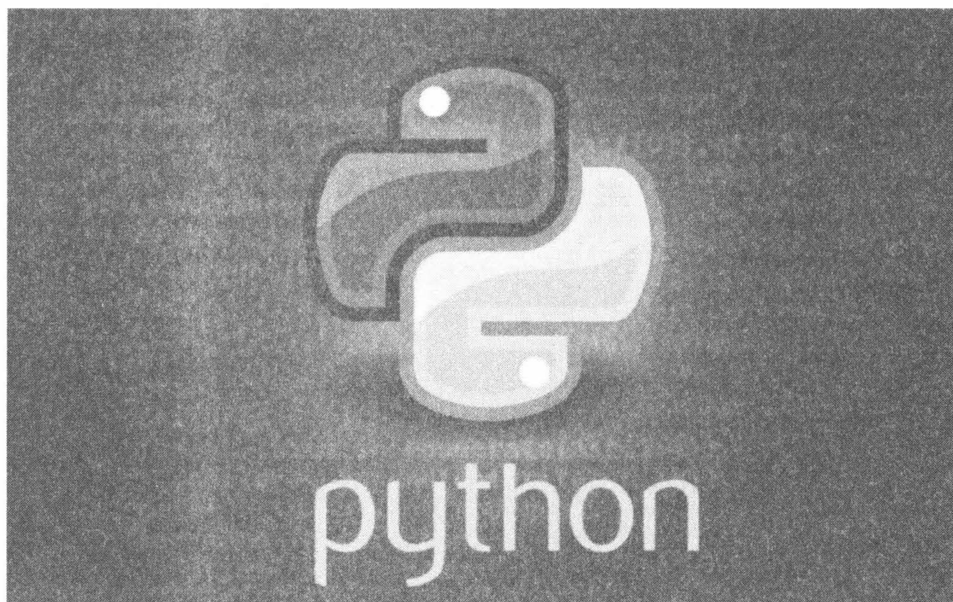
Функция *itertools.combinations_with_replacement()* позволяет одному и тому же элементу выбираться несколько раз. Например:

```
>>> for c in combinations_with_replacement(items, 3):
...     print(c)
...
('a', 'a', 'a')
('a', 'a', 'b')
('a', 'a', 'c')
('a', 'b', 'b')
('a', 'b', 'c')
('a', 'c', 'c')
('b', 'b', 'b')
('b', 'b', 'c')
('b', 'c', 'c')
('c', 'c', 'c')
>>>
```

Этот пример демонстрирует только часть всей мощи, которую вы можете обнаружить в модуле **itertools**. Несмотря на то, что вы можете, конечно, записать код, чтобы произвести перестановки сами, это потребует дополнительных затрат времени. Да и зачем изобретать колесо заново? Когда вы сталкиваетесь с итеративными задачами, в первую очередь обратитесь к модулю **itertools**. Если ваша задача распространенная, вполне возможно, что ее решение уже есть в **itertools**.

ГЛАВА 19.

ДОКУМЕНТИРОВАНИЕ ПРОЕКТА



Очень часто документированию проекта уделяется мало внимания или же вообще этим никто не занимается. Делается это по самым разным причинам, среди которых можно выделить нехватку времени, финансирования, отсутствие технических писателей в штате. Почему-то так повелось, что никто не закладывает в проект время/финансы, необходимое на разработку документации – ни разработчик, ни заказчик.

Заказчик часто не хочет тратить дополнительные средства в надежде, что в случае чего есть разработчик, который все поправит. Но в жизни все меняется – компании, которая разрабатывала проект, может уже и не быть на момент необходимости внесения изменений, отношения с этой компанией могут быть испорчены и т.д. Заказчик почему-то думает, что сторонним разработчикам будет просто разобраться в коде, который был написан кем-то другим. На практике бывает так, что даже самому разработчику несколько месяцев спустя сложно разобраться в собственном коде и нужно время, чтобы все вспомнить. Наличие документации позволит "вспомнить все" гораздо быстрее, а также разобраться в коде сторонним разработчикам, если в этом возникнет необходимость.

19.1. Рекомендации относительно написания технической документации

Поддерживать хорошую документацию довольно несложно, даже проще, чем писать код, хотя многие думают иначе. Документировать проект станет проще, если придерживаться некоторых рекомендаций относительно технического письма.

Если вы никогда раньше ничего не писали, не отчаивайтесь – ведь вам нужно написать не роман, а всего лишь описать, как работает ваши API (*Application Programming Interface, API* — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой). Грубо говоря, составить список функций, описать назначение каждой функции и назначение каждого передаваемого в функцию параметра.

Вкратце рекомендации выглядят так:

- Сформулируйте направления, а затем развивайте их. Первым делом опишите список того, о чем вы хотите написать, если хотите – оглавление. Дальше уже подробно описывайте каждый из разделов.
- Помните о читателе. При написании документации помните, что ее кто-то будет читать. Если вы пишите руководство по панели управления (*dashboard*), тогда нужно ориентироваться на обычного пользователя и, возможно, некоторые моменты нужно описать подробнее. Если вы описываете API, тогда нужно ориентироваться на технического специалиста.
- Краткость – сестра таланта. Не нужно писать книгу, а тем более книгу в нескольких томах. Пишите кратко и по делу.
- Используйте шаблоны. Шаблоны помогают читателю привыкнуть к общей структуре документов.
- Используйте реальные примеры кода. Вместо того, чтобы фантазировать и высказывать предположения, демонстрируйте реальные примеры кода и объясняйте, как он работает.

19.1.1. СФОРМУЛИРУЙТЕ НАПРАВЛЕНИЯ, А ЗАТЕМ РАЗВИВАЙТЕ ИХ

Невозможно написать хороший текст за один проход. Не нужно пытаться придумать сразу же какой-то идеальный и всеобъемлющий текст. Вместо этого набросайте идеи, а затем развивайте их. Написав пару предложений, обязательно перечитайте их. Возможно, придется внести корректировки.

План текста обязательно должен быть. В голове все удержать невозможно – кто-то позвонил, написал, вы отвлеклись и уже забыли, о чем думали.

Есть и второй подход. Излагайте все идеи на бумаге, начинайте записывать непрерывный поток мыслей и не останавливайтесь, не обращайте внимания ни на стиль подачи, ни на грамматически ошибки/опечатки, просто пишите. Не останавливайтесь, пока не изложите все идеи. А уже после всего это займитесь структурированием текста, его улучшением и т.д.

19.1.2. ПОМНИТЕ О ЧИТАТЕЛЕ

При написании документа или его части вы должны понимать, кто будет вашей целевой аудиторией. Это очень важно.

Не нужно стараться написать техническую документацию так, чтобы она была понятной даже школьнику. Так не должно быть. С техническим текстом все должно быть проще: он должен предназначаться определенному типу читателя. Придерживаясь этого правила, вы сэкономите кучу времени.

Если вы пишете документацию по API, то ваш единственный тип читателя – программист. Ориентируйтесь на него. Если пишете документацию по панели управления – здесь вашим читателем будет обычный среднестатистический пользователь, об этом тоже нужно думать. Не нужно беспокоиться о том, что этот текст может быть прочитан начальством со стороны заказчика. Ведь начальство – это такой же обычный пользователь, как и все остальные.

Создайте два разных документа – один для пользователя (например, dashboard.pdf), второй – для разработчика (api.pdf). В каждом документе придерживайтесь собственного стиля подачи материала – так чтобы текст был понятен целевой аудитории. Понятно, что можно написать api.pdf так, что он станет понятен даже пользователю, но зачем? Пользователь читать

его не будет, поскольку это не его обязанности, а у программиста лишь вызовет улыбку. Вы только потратите время, а лучше никому не сделаете.

19.1.3. КРАТКОСТЬ – СЕСТРА ТАЛАНТА

Упрощайте все по возможности. Простой текст воспринимается проще. Пишите простые и короткие предложения. Даже если вы не обладаете талантом писателя, ваше творчество будет проще понять другим людям. Вы ведь пишите не роман и не сказку, а руководство по использованию API вашего программного продукта. Не нужно, чтобы это руководство (без крайней на то необходимости) занимало объем в 1000 страниц.

Придерживайтесь следующих рекомендаций:

1. Пишите короткие предложения. Максимальная длина предложения – 150 символов с пробелами. В среднем в одной строке листа формата А4 помещается 80-85 символов с пробелами. Так что старайтесь, чтобы ваше предложение не занимало более двух строк.
2. Каждый абзац должен выражать свою идею и состоять из 3-4 предложений. Не нужно писать абзацы, состоящие из более чем 10 предложений. 10 предложений по 150 символов каждое – более чем достаточно для выражения законченной идеи.
3. Не используйте разные формы времени, при написании технической документации достаточно настоящего времени.
4. Избегайте шуток, вы пишите не роман и даже не книгу по программированию. Вы просто пишите техническую документацию. Шутки и прочий юмор в ней смотрится глупо и неуместно.
5. Не повторяйте одни и те же слова множество раз. Используйте аббревиатуры. Например, вместо слова "программное обеспечение" пишите ПО. Как правило, в первый раз аббревиатура расшифровывается (ПО (программное обеспечение) или программное обеспечение (далее – ПО)), далее используется без расшифровки. Можно также составить список используемых аббревиатур, если их слишком много.

Плохая документация имеет одну отличительную черту – в ней невозможно найти нужную информацию, даже если она там есть. Продумайте структуру документа, разбивайте текст на разделы, используйте осмысленные

названия заголовков. Эти простые правила помогут создать понятный документ, который будет просто читать. Если же всего этого вы не будете придерживаться, читатели будут использовать средства поиска, чтобы найти нужную информацию. Но когда документация находится в распечатанном виде, сделать это будет очень проблематично.

19.1.4. ИСПОЛЬЗУЙТЕ ШАБЛОНЫ

В качестве примера использования шаблонов можно привести всем известную "Википедию". Все страницы "Википедии" выглядят одинаково. Справа находятся блоки, обобщающие информацию из документов, которые принадлежат к одной и той же области. Первая часть статьи обычно представляет собой оглавление со ссылками, которые ведут на якоря в этом же тексте. А в конце странички всегда есть справочная информация.

Шаблоны – штука полезная. Во-первых, к ним привыкают читатели. Они знают, что в каждом документе/разделе есть оглавление, позволяющее быстро найти нужную информацию. Также они знают, что внизу документа обязательно будет справочная информация со ссылками на полезные темы. Пользователи привыкают к общей структуре информации и учатся читать быстрее.

Во-вторых, шаблоны полезны и для самого технического писателя – это быстрый старт для нового документа. Вам не нужно ничего выдумывать, вы просто используете уже готовый шаблон.

19.1.5. ИСПОЛЬЗУЙТЕ РЕАЛЬНЫЕ ПРИМЕРЫ КОДА

Помните, что вы пишите не учебник, объясняющий, как программировать на том языке, на котором написан ваш программный продукт, а техническую документацию по программному продукту. Следовательно, нереалистичные примеры кода усложняют понимание такой документации.

Проще всего взять реальный фрагмент кода и пояснить все на нем. Рассмотрим небольшой пример:

```
# Получаем e-mail пользователя из объекта user
```

```
uemail = user.email;

# запрашиваем количество доступных уроков из CRM
  response = requests.get("http://example.crm/alfa/alfa.
php?email=" + uemail + "&op=getlessons")

# Текст ответа - это и есть количество доступных уроков
  lessons = response.text
# Получаем дату регистрации пользователя
  join_date = user.date_joined
# Задаем граничную дату регистрации, начиная с этой даты
# все зарегистрированные пользователи будут считаться новыми
  border_date = dateutil.parser.parse('2021-09-23
00:00:00+00:00')

# если граничная дата "старше" даты регистрации, то
пользователь - старый
# иначе - пользователь новый и переменную new_user нужно
установить в 1
  if border_date > user.date_joined:
    new_user = 0
  else:
    new_user = 1

# Флаг сокрытия курсов
# Если пользователь новый и количество уроков = 0, то скрываем
меню выбора курса
  hide_courses = 0
  if new_user == 1  and lessons == "0":
    hide_courses = 1
```

Мы только что не просто привели фрагмент кода, но и в комментариях подробно объяснили, для чего он используется.

19.2. Строки документации в Python

Лучший способ поддерживать документацию в актуальном состоянии – относиться к ней, как к коду. Храните в репозитарии, отслеживайте вносимые изменения – так легко поддерживать актуальность документации.

Используя систему управления версиями (тот же Git), вы с легкостью сможете отследить все изменения, внесенные в документацию – не только вами, но и другими членами команды.

Существуют инструменты, позволяющие создавать документацию API прямо из комментариев, которые включены в исходный код. Это считается лучшим способом создания технической документации для библиотеки или любого другого компонента, подразумевающего многократное использование кода.

Что касается Python, то у него есть уникальные качества, облегчающие процесс документирования. Вы можете создавать красивую и полезную документацию прямо из кода Python. Основной для этих инструментов служат строки документации (*docstrings*).

Строки документации представляют собой специальные строковые литералы Python, предназначенные для документирования функций, методов, классов и моделей Python. Если первое определение функции, метода, класса или модуля является строковым литералом, то он автоматически станет строкой документации и превратится в значение атрибута __doc__ для этой функции, метода, класса или модуля.

Строки документации должны иметь все модули, все функции и классы, экспортируемые модулем. Также должны иметь строки документации публичные методы (в том числе __init__).

Для описания строк документации всегда используются три двойных кавычки. Например: """строка документации""". Существует две формы строк документации: однострочная и многострочная.

Однострочные строки документации должны помещаться на одной строке, и использоваться для очевидных случаев:

```
def get_user_info():
   """Возвращает массив с информацией о пользователе"""
   global _user
   if _user: return _user
```

Используйте тройные кавычки, даже если документация умещается на одной строке. Закрывающие кавычки размещайте на той же строке. Это смотрится лучше.

Многострочные строки документации состоят из однострочной строки документации с последующей пустой строкой, а затем более подробным опи-

санием. Первая строка может быть использована автоматическими средствами индексации, поэтому важно, чтобы она находилась на одной строке, и была отделена от остальной документации пустой строкой. Первая строка может быть на той же строке, где и открывающие кавычки, или на следующей строке. Вся документация должна иметь такой же отступ, как кавычки на первой строке. Рассмотрим пример:

```
def sum(a=0.0, b=0.0):
    """Вычисляет сумму двух чисел

    Аргументы
    a - первое число (по умолчанию 0.0)
    b - второе число (по умолчанию 0.0)

    """
    return a+b
    ...
```

19.3. Языки разметки для документации

Внутри строк документации можно писать все, что угодно. Но желательно использовать какой-то язык разметки для документации, что поможет вам в дальнейшем. Неплохими вариантами являются языки разметки Markdown и AsciiDoc. Первый очень популярен в сообществе GitHub и является наиболее распространенным языком разметки. Также он поддерживается различными инструментами для самодокументирующихся API.

Вообще, вы можете выбрать любой язык разметки, на котором вам легко писать и какой удобно читать в сыром виде за пределами сгенерированной документации. Неплохим вариантом также является язык разметки reStructured Text, который используется системой документации Sphinx.

19.4. Популярные генераторы документации

Наиболее популярными инструментами для создания документации в сообществе Python являются Sphinx и MkDoks. Сначала мы рассмотрим Sphinx, а затем – MkDocs.

19.4.1. ИСПОЛЬЗОВАНИЕ SPHINX

Sphinx (англ. *SQL Phrase Index*) — система полнотекстового поиска, отличительной особенностью которого является высокая скорость индексации и поиска, а также интеграция с существующими СУБД (MySQL, PostgreSQL) и API для распространенных языков веб-программирования.

Для установки Sphinx введите команду:

```
pip install sphinx
```

> ***Примечание.*** Если команда pip недоступна, ее установить можно следующей командой sudo apt install python3-pip

В Ubuntu его можно также установить командой:

```
sudo apt-get install python3-sphinx
```

Использовать инструмент не очень сложно. Первым делом, нужно создать каталог для нового проекта и перейти в него:

```
mkdir prj
cd prj
```

Для инициализации проекта необходимо выполнить команду *sphinx-quickstart*:

```
sphinx-quickstart
```

Программа задаст ряд вопросов. Все настройки можно будет позже изменить в файле conf.py.

```
> Корневой каталог документации. По умолчанию текущий каталог.
> Root path for the documentation [.]:

> Сделать ли раздельные папки исх. кода и готовых страниц - Да
> Separate source and build directories (y/N) [n]: y
```

```
> Префикс для директорий с шаблонами и статическими файлами.
> Name prefix for templates and static dir [_]:

> Название проекта. Для начала лучше вводить на латинице.
> Project name:

> Имя автора/авторов. Для начала лучше вводить на латинице.
> Author name(s):

> Версия проекта
> Project version:

> Номер релиза проекта
> Project release [1]:

> Расширение исходного файла. По умолчанию .rst.
> Source file suffix [.rst]:

> Имя мастер-документа. По умолчанию index.rst.
> Name of your master document (without suffix) [index]:

> Генерировать ePub версию документации?
> Do you want to use the epub builder (y/n) [n]:

> Автоматически вставлять docstrings из модулей
> autodoc: automatically insert docstrings from modules (y/n) [n]:

>
> doctest: automatically test code snippets in doctest blocks (y/n)
[n]:

>
> intersphinx: link between Sphinx documentation of different
projects (y/n) [n]:

>
> todo: write "todo" entries that can be shown or hidden on build
(y/n) [n]:

>
> coverage: checks for documentation coverage (y/n) [n]:

> Использовать модуль pngmath для вставки формул в формате png
> pngmath: include math, rendered as PNG images (y/n) [n]:

> Использовать модуль mathjax для вставки формул в формате MathJax
> mathjax: include math, rendered in the browser by MathJax (y/n)
[n]: y
```

```
>
> ifconfig: conditional inclusion of content based on config values
(y/n) [n]:

>
> viewcode: include links to the source code of documented Python
objects (y/n) [n]:

> Создать Makefile - да
> Create Makefile? (y/n) [y]:

> Сделать ли файл .bat, - нет, если у вас Linux
> Create Windows command file? (Y/n) [y]: n ()
```

После ответа на вопросы будут созданы файлы index.rst, conf.py, Makefile, _build, _static, _templates:

- Makefile — содержит инструкции для генерации результирующего документа командой make.
- _build — директория, в которую будут помещены файлы в определенном формате после того, как будет запущен процесс их генерации.
- _static — в эту директорию помещаются все файлы, не являющиеся исходным кодом (например, изображения). Позже создаются связи этих файлов с директорией build.
- conf.py — содержит конфигурационные параметры Sphinx, включая те, которые были выбраны при запуске sphinx-quickstart в окне терминала.
- index.rst — это корень проекта. Он соединяет документацию воедино, если она разделена на несколько файлов.

Файл index.rst используется для объединения всех файлов в один проект. По сути, это простой текстовый файл. Если открыть его, то можно будет увидеть примерно следующее:

```
.. 3 documentation master file, created by
sphinx-quickstart on Fri Sep 24 19:44:30 2021.
You can adapt this file completely to your liking, but it
should at least
contain the root `toctree` directive.
```

```
Welcome to 3's documentation!
=============================

Contents:

.. toctree::
   :maxdepth: 2


Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Содержимое index.rst не должно включать много информации и в нем обязательно должна присутствовать директива .. toctree::. Чтобы включить в проект другие файлы, необходимо прописать названия этих файлов в .. toctree::. Пример:

```
.. toctree::
   :maxdepth: 2

   example1
   example2
```

Здесь мы включили в проект два файла – example1.rst и example2.rst. Обратите внимание, что название файла пишется без расширения. Также важен отступ и пустая строка.

Для генерации документации в HTML формат необходимо выполнить в командной строке команду *make html*. Аналогичным образом можно выполнить генерацию в другие форматы, например, *make epub*.

```
cd prj
make html
```

Произойдет сборка HTML, выходные файлы будут помещены в директорию _build/html/. Перейдите в нее и откройте файл index.html в браузере.

Получив совсем немного исходных данных, Sphinx сумел создать нечто большее. Вы получите несложную компоновку, содержащую информацию о документации проекта, раздел поиска, содержание, заметки об авторских правах, включая имя и дату, а также нумерацию страниц. И все это сделано автоматически!

19.4.2. ИСПОЛЬЗОВАНИЕ MKDOCS

Оба генератора документации, и Sphinx, и MkDocs написаны на Python. Sphinx известен тем, что именно на нем сгенерирована документация для языка Python. MkDocs — это бесплатный статический генератор сайтов, предназначенный для создания проектной документации. Его можно использовать для создания автономного сайта с документацией по программному продукту.

Так как MkDocs создает статические файлы, ваша документация является легкой и простой в размещении — с использованием бесплатных сервисов, таких как GitHub Pages и Read The Docs, — или, конечно, на вашем собственном сервере. Все, что вам нужно – это просто скопировать получившиеся файлы в каталог на веб-сервере.

Для установки инструмента введите команду:

```
pip install mkdocs
```

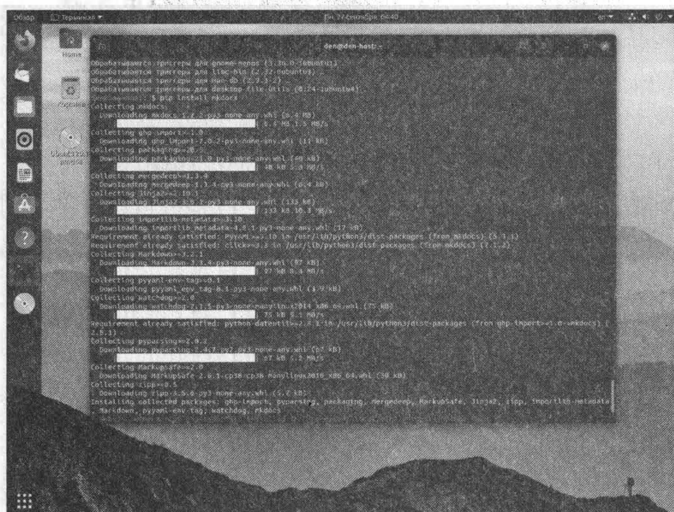


Рис. 19.1. Установка mkdocs

В Ubuntu mkdocs можно установить командой:

```
sudo apt install mkdocs
```

После этого нужно создать новый проект и перейти в его каталог:

```
cd ~
mkdocs new prj
cd prj
```

Вывод второй команды будет таким:

```
INFO - Creating project directory: prj
INFO - Writing config file: prj/mkdocs.yml
INFO - Writing initial docs: prj/docs/index.md
```

После этого в каталоге **prj** будут созданы следующие элементы:

- файл конфигурации mkdocs.yml
- одна страница документации docs/index.md

Далее запустим dev-сервер для просмотра вашей документации. Убедитесь, что вы находитесь в том же каталоге, что и файл mkdocs.yml, и введите следующую команду:

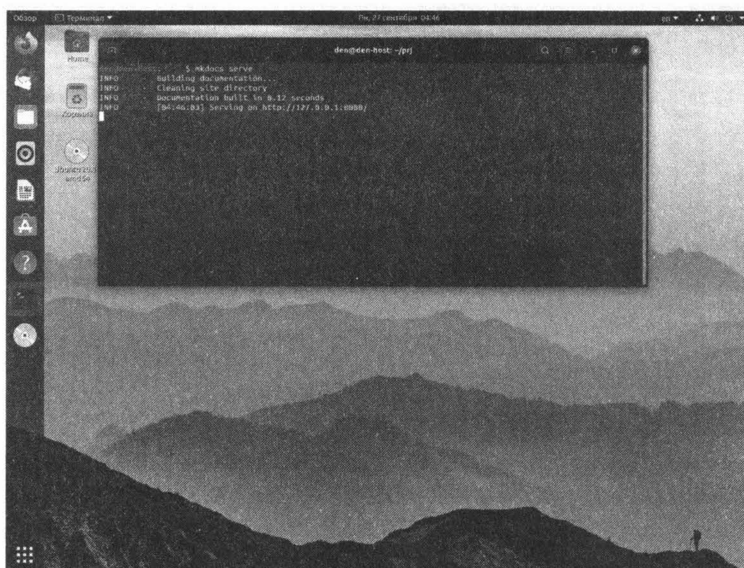


Рис. 19.2. Запуск веб-сервера

```
$ mkdocs serve
INFO    -  Building documentation...
INFO    -  Cleaning site directory
[I 270921 03:50:43 server:271] Serving on
http://127.0.0.1:8000
[I 270921 03:50:43 handlers:58] Start watching changes
[I 270921 03:50:43 handlers:60] Start detecting changes
```

Откройте браузер и введите URL *http://127.0.0.1:8000*. Вы увидите сгенерированный шаблон для вашей будущей документации.

Начало положено. Далее нужно открыть mkdocs.yml и указать название вашего проекта и адрес сайта проекта:

```
site_name: MyPrj
site_url: https://example.com/
```

Перезагрузите страничку в браузере, и вы сразу же увидите изменения – вместо MkDocs будет отображаться название вашего проекта (MyPrj).

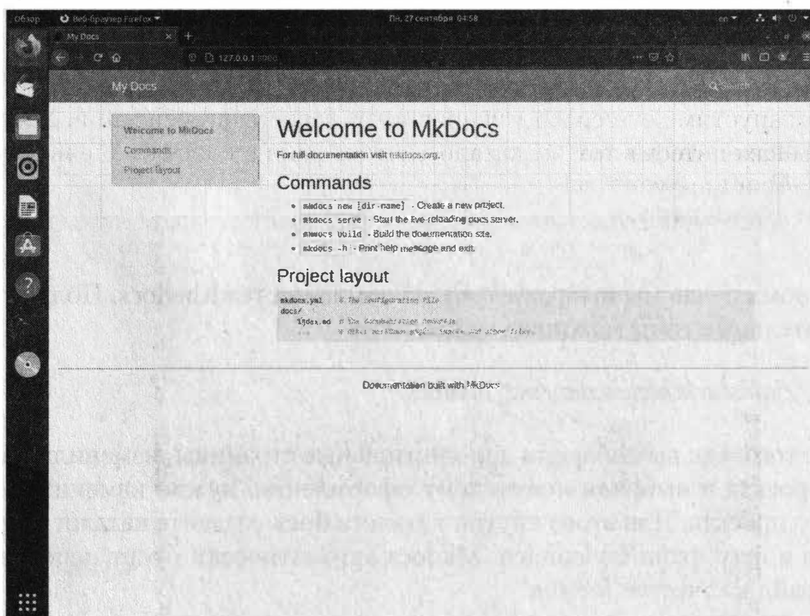


Рис. 19.3. Сгенерированная документация

Настало время добавить в наш проект дополнительные страницы. Для этого в mkdocs.yml нужно добавить секцию *nav*:

```
nav:
    - Home: index.md
    - About: about.md
```

Страничка index.md уже создана, а вот страничка docs/about.md отсутствует. Создадим ее так:

```
curl 'https://jaspervdj.be/lorem-markdownum/markdown.txt' >
docs/about.md
```

Для изменения темы оформления документации используется параметр **theme** в файле конфигурации:

```
site_name: MyPrj
site_url: https://example.com/
nav:
    - Home: index.md
    - About: about.md
theme: readthedocs
```

В данном случае мы выбрали тему оформления readthedocs. Получить дополнительные темы можно на сайте:

https://jamstackthemes.dev/ssg/mkdocs/

После того, как вы добавили дополнительные страницы, изменили параметры проекта и выбрали новую тему оформления, нужно изменить *favicon* вашего проекта. Для этого внутри каталога **docs** создайте каталог **img** и добавьте в него файл favicon.ico. Mkdocs автоматически будет использовать этот файл в качестве *favicon*.

Когда результат в браузере по адресу 127.0.0.1:8000 вас будет устраивать, настанет время для сборки сайта с документацией. Перейдите в каталог **prj** и введите команду:

```
mkdocs build
```

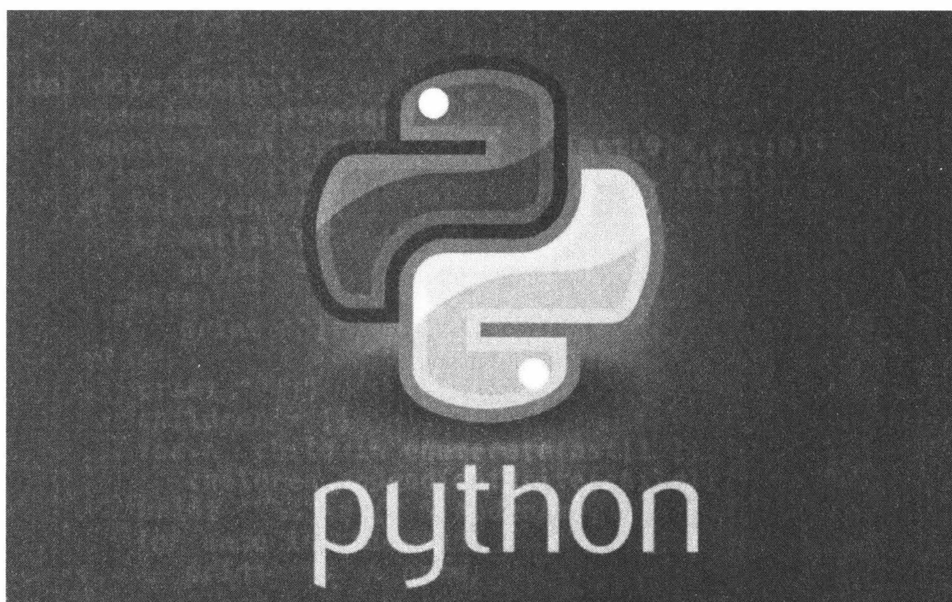
Внутри каталога проекта будет создан подкаталог **site**, в котором будет находиться HTML-версия вашей документации. Содержимое этого каталога можно опубликовать в Интернете, или поставлять вместе с вашим программным продуктом.

Нужно отметить, что представленные генераторы документации, которые хоть и рассмотрены в контексте Python, можно использовать для создания документации по любому программному продукту, независимо от языка программирования, на котором он написан.

https://www.mkdocs.org/getting-started/

ГЛАВА 20.

МЕТА-ПРОГРАММИРОВАНИЕ



Метаклассы – это магия, о которой 99% пользователей не стоит даже задумываться. Если вам интересно, нужны ли они вам – тогда точно нет. Люди, которым они на самом деле нужны, знают, зачем, и что с ними делать.

~ Гуру Python Tim Peters

20.1. Введение в метапрограммирование

Метапрограммирование – подход к написанию программ, которые можно обрабатывать как данные, что позволяет им просматривать, создавать/изменять себя во время работы. Другими словами, мы можем написать программу, которая будет изменять сама себя.

В основном, метаклассы используются для создания API. Типичным примером является Django ORM. Можно написать что-то вроде этого:

```
class Person(models.Model):
    name = models.CharField(max_length=30)
    age = models.IntegerField()
```

Но если написать так:

```
user = Person(name='den', age='38')
print(user.age)
```

Он не вернет объект IntegerField. Он вернет код *int* и даже может взять его непосредственно из базы данных.

Существует два основных подхода к метапрограммированию:

- Первый концентрируется на возможности языка анализировать собственные элементы – функции, классы, типы, а также на возможности создавать или изменять их динамически, то есть в процессе выполнения программы. В Python есть множество инструментов для этого. Эти инструменты используются различными IDE для анализа кода в режиме реального времени. Также существуют специальные методы классов, позволяющие вмешиваться в процедуру создания экземпляра класса – метаклассы. Благодаря метаклассам программист может полностью переделать реализацию ООП в Python.
- Второй подход заключается в возможности программиста работать непосредственно с кодом – как в простом виде (обычный текст), так и в форме абстрактного синтаксического дерева (AST, Abstract Syntax, Tree). Такой подход более сложный в реализации, зато дает возможность делать очень сложные и эффектные штуки, такие как расширение синтаксиса самого Python или даже создание собственного языка программирования.

Далее мы рассмотрим декораторы в контексте метапрограммирования.

20.2. Декораторы

Декораторы – это синтаксический сахар, работающий по простой схеме:

```
def decorated_functioin():
  pass
decorated_function = some_decorator(decorated_function)
```

Данная форма показывает, что именно делает декоратор: он принимает объект функции и изменяет его во время выполнения. Другими словами, новая

функция создается на основе предыдущей версии функции с тем же именем. Такое декорирование может быть очень сложной операцией, которая выполняет некоторый самоанализ кода (а не такой простой случай, как мы рассмотрели). Однако, все это позволяет использовать декораторы в качестве инструмента метапрограммирования.

Основные принципы декораторов довольно просты, и это очень хорошо, поскольку остальные способы метапрограммирования в Python гораздо сложнее и приводят к существенному усложнению кода.

В Python программист может использовать декораторы класса. По синтаксису и реализации они аналогичны декораторам функций.

Декоратор класса призван модифицировать поведение и содержание класса, не изменяя его исходный код. Похоже на наследование, но есть отличия:

1. Декоратор класса имеет более глубокие возможности по влиянию на класс, он может удалять, добавлять, менять, переименовывать атрибуты и методы класса. Он может возвращать совершенно другой класс.
2. Старый класс "затирается" и не может быть использован, как базовый класс при полиморфизме
3. Декорировать можно любой класс одним и тем же универсальным декоратором, а при наследовании – мы ограничены иерархией классов и должны считаться с интерфейсами базовых классов.
4. Презираются все принципы и ограничения ООП (из-за пунктов 1-3).

Декораторы классов полезны, чтобы внедриться в класс и массово воздействовать на его методы и атрибуты. Далее мы создадим декоратор, который будет измерять время выполнения каждого метода класса. При этом сам класс никаких изменений не претерпит, и не будет знать, что за ним следят:

Листинг 20.1. Декоратор класса

```
import time

# это вспомогательный декоратор будет декорировать каждый
метод класса,
# см. далее
def timeit(method):
    def timed(*args, **kw):
        ts = time.time()
        result = method(*args, **kw)
        te = time.time()
        delta = (te - ts) * 1000
        print(f'{method.__name__} выполнялся {delta:2.2f} ms')
```

```
        return result
    return timed


def timeit_all_methods(cls):
    class NewCls:
        def __init__(self, *args, **kwargs):
            # проксируем полностью создание класса
            # создаем декорируемый класс
            self._obj = cls(*args, **kwargs)

        def __getattribute__(self, s):
            try:
                # есть ли атрибут s?
                x = super().__getattribute__(s)
            except AttributeError:
                # такого атрибута нет
                pass
            else:
                # такой атрибут есть
                return x

            # если объект, значит, должен быть атрибут s
            attr = self._obj.__getattribute__(s)

            # метод ли он?
            if isinstance(attr, type(self.__init__)):
                # да, обернуть его в измеритель времени
                return timeit(attr)
            else:
                # не метод, что-то другое
                return attr
    return NewCls


@time_all_class_methods
class Foo:
    def a(self):
        print("метод a начинает работу")
        time.sleep(0.888)
        print("метод a завершает работу")


f = Foo()
f.a()
```

Вывод будет таким:

```
# метод a начинает работу
# метод a завершает работу
# a 889.84 ms
```

Разберемся, что и к чему. Наш измеритель времени, *timeit* – простой декоратор для функций. Он нам нужен, чтобы декоратор класса *timeit_all_methods* обернул в *timeit* каждый метод декорируемого класса.

Декоратор *timeit_all_methods* содержит в себе определение нового класса NewCls и возвращает его вместо оригинального класса. Другими словами, класс **Foo** – это уже не Foo, а NewCls. Конструктор класса NewCls принимает произвольные аргументы (ведь нам не известно заранее, какой конструктор у Foo, и у любого другого класса, который мы декорируем). Поэтому конструктор просто создает поле, где будет хранить экземпляр оригинального класса, и передает ему в конструктор все свои аргументы.

Метод _getattribute_ вызывается, когда что-то пытается обратиться к какому-то атрибуту (полю, методу) класса NewCls. Мы должны обратиться к родителю *super()* и спросить у него, не обладаем ли мы сами атрибутом, который проверяем. Нужно обращаться именно к родителю, чтобы избежать рекурсии. Если это наш *атрибут* (*атрибут класса декоратора*) – вернем его сразу, с ним ничего не надо делать. В противном случае – если это атрибут исходного класса – нужно запросить его у него. Далее нужно проверить его тип, сравним его с типом любого метода. Если тип – *метод* (*bound method*), то обернем его в декоратор *timeit* и вернем, иначе (это не метод, а свойство или статический метод) – вернем без изменений.

В декораторах классов также доступны замыкания и параметризация. Пользуясь данными фактами, предыдущий пример можно переписать, сделав более читабельным и удобным в сопровождении:

```
def parametrized_short_mtd(max_width=8):
  """Параметризованный декоратор, сокращающий представление"""
  def parametrized(cls):
    """Внутренняя функция-оболочка, которая по сути является
декоратором"""
    class ShortlyRepresented(cls):
      """Подкласс, представляющий поведение декоратора"""
      def __mtd__(self):
        return super().__mtd__()[:max_width]
```

```
    return ShortlyRepresented

  return parametrized
```

Главный недостаток использования замыканий в декораторах классов в том, что полученные объекты являются не экземплярами класса, который был задекорирован, а экземплярами подкласса, созданного динамически в функции декоратора. Среди прочего, это повлияет на атрибуты __name__ и __doc__:

```
@parametrized_short_mtd(10)
class ClassWithLittleBitLongerLongName:
  pass
```

Такое использование декораторов класса приведет к следующим изменениям в метаданных класса:

```
>>> ClassWithLittleBitLongerLongName().__class__
<class 'ShortlyRepresented'>
>>> ClassWithLittleBitLongerLongName().__doc__
'Подкласс, представляющий поведение декоратора'
```

20.3. Метаклассы

20.3.1. ВВЕДЕНИЕ В МЕТАКЛАССЫ

Метаклассы – одна из самых трудных концепций в Python, поэтому многие программисты избегают ее использования. Давайте разберемся, что такое метакласс и как его можно использовать для метапрограммирования.

Метакласс – это тип (класс), который определяет другие типы (классы). Грубо говоря, это "фича", создающая новые классы. Классы, определяющие экземпляры объектов, также являются объектами. А поэтому у них есть соответствующий класс. Основным типом каждого класса по умолчанию является встроенный класс **type**.

Рассмотрим общий синтаксис метаклассов. Мы можем использовать вызов встроенного класса *type()* в качестве динамического объявления класса. Например, мы можем определить класс вызовом *type()*:

```
def method(self):
  return 1

MyCls = type('MyCls', (object,), {'method': method})
```

Все это эквивалентно обычному определению класса ключевым словом *Class*:

```
class MyCls:
  def method(self):
    return 1
```

У каждого класса, создаваемого таким образом, есть метакласс **type**. Такое поведение по умолчанию можно изменить, если добавить именованный аргумент **metaclass**:

```
class ClMeta(metaclass=type):
  pass
```

Здесь значение, предоставляемое в качестве атрибута **metaclass**, – еще один объект класса, но может быть любым другим вызываемым объектом, который принимает те же аргументы, что и класс **type** и возвращает другой объект класса.

Рассмотрим аргументы **type**:

- *name* – имя класса, которое будет храниться в атрибуте __name__;
- *bases* – список родительских классов, которые станут атрибутом __base__;
- *dict* – словарь, который будет являться пространством имен для тела класса (становится атрибутом '__dict__').

Почему имя *type()* пишется со строчной буквы? Скорее всего, это вопрос соответствия со *str* – классом, который отвечает за создание строк, и *int* – классом, создающим целочисленные объекты. **type** – это просто класс, создающий объекты класса. Проверить можно с помощью атрибута __class__. Все, что вы видите в Python – объекты. В том числе и строки, числа, классы и функции. Все это объекты, и все они были созданы из класса:

```
>>> name = 'den'
>>> name.__class__
<type 'str'>
>>> age = 38
>>> age.__class__
<type 'int'>
>>> def foo(): pass
>>> foo.__class__
<type 'function'>
>>> class Cls(object): pass
>>> c = cls()
>>> c.__class__
<class '__main__.Cls'>
```

А теперь самое интересное – что в атрибуте __class__ у самого __class__?

```
>>> name.__class__.__class__
<type 'type'>
>>> age.__class__.__class__
<type 'type'>
>>> foo.__class__.__class__
<type 'type'>
>>> c.__class__.__class__
<type 'type'>
```

Итак, метакласс создает объекты класса. Это можно назвать "фабрикой классов". **type** – встроенный метакласс, который использует Python. Программист может также создать свой собственный метакласс.

Вернемся к атрибуту __metaclass__. При определении класса вы можете указать этот атрибут:

```
class Foo(object):
    __metaclass__ = something...
    [...]
```

После этого Python будет использовать метакласс для создания класса Foo.

Если написать class Foo(object), объект класса Foo не сразу создастся в памяти – Python будет искать __metaclass__. Как только атрибут будет найден, он будет использоваться для создания класса Foo. В том случае, если этого не произойдет, Python будет использовать **type** для создания класса.

Рассмотрим пример:

```
class Foo(Bar):
   pass
```

При таком объявлении Python проверит, есть ли атрибут __metaclass__ у класса Foo? Если он есть, создаст в памяти объект класса с именем Foo с использованием того, что находится в __metaclass__. Если __metaclass__ не найден, то Python будет искать его на уровне модуля и после этого повторит процедуру. В случае если он вообще не может найти какой-либо __metaclass__, Python использует собственный метакласс **type**, чтобы создать объект класса.

Наверное, вас интересует вопрос: что можно добавить в __metaclass__. Да практически все, что может создавать классы. Как минимум **type** или его подклассы, а также все, что использует **type**.

20.3.2. ПОЛЬЗОВАТЕЛЬСКИЕ МЕТАКЛАССЫ

Основная цель метакласса – автоматическое изменение класса во время его создания. Обычно это делается для API, когда нужно создать классы, соответствующие текущему контексту. Например, нам нужно, чтобы все классы в модуле должны иметь свои атрибуты, и они должны быть записаны в верхнем регистре. Чтобы решить эту задачу, можно задать __metaclass__ на уровне модуля.

Просто нужно сообщить метаклассу, что все атрибуты должны быть в верхнем регистре. __metaclass__ действительно может быть любым вызываемым объектом, он не обязательно должен быть формальным классом.

Начнем с очень простого примера с использованием функции:

```
# метаклассу автоматически передается тот же аргумент,
# который вы обычно передаете в `type`
def upper_attr(future_class_name, future_class_parents,
future_class_attr):
    """
      Возвращает объект класса со списком его атрибутов
      в верхнем регистре
```

```
    """

    # выбирает любой атрибут, который не начинается с "_" и
переводит его
    # в верхний регистр
    uppercase_attr = {}
    for name, val in future_class_attr.items():
        if not name.startswith('__'):
            uppercase_attr[name.upper()] = val
        else:
            uppercase_attr[name] = val

    # type создаст класс
    return type(future_class_name, future_class_parents,
uppercase_attr)

__metaclass__ = upper_attr # это повлияет на все классы в
модуле

# Глобальный _metaclass__ не будет работать с "объектом", но мы можем
# определить здесь __metaclass__, чтобы воздействовать только на этот
# класс, и он будет работать с дочерними элементами "объекта"
class Foo():
    bar = 'bip'

print(hasattr(Foo, 'bar'))
# Выводит: False
print(hasattr(Foo, 'BAR'))
# Выводит: True

f = Foo()
print(f.BAR)
# Выводит: 'bip'
```

Теперь проделаем то же самое, но с использованием метакласса:

```
# помните, что `type` - это такой же класс, как `str` и `int`
# поэтому вы можете наследовать его
class UpperAttrMetaclass(type):
    # __new__ - это метод, который вызывается до __init__
    # он создает объект и возвращает его
    # а метод __init__ просто инициализирует объект, переданный как параметр
    # вам нужно редко использовать __new__, кроме случаев, когда вы хотите
    # контролировать создание объекта
    # В этом примере создаваемым объектом является класс и мы хотим
    # кастомизировать его, поэтому мы переопределяем __new__
```

```
    # Вы можете сделать некотору работу в __init__ если вам это нужно
    # Некоторые особо продвинутые программисты переопределяют метод __call__
    # но мы не будем этого делать
    def __new__(upperattr_metaclass, future_class_name,
                future_class_parents, future_class_attr):

        uppercase_attr = {}
        for name, val in future_class_attr.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

        return type(future_class_name, future_class_parents, uppercase_attr)
```

Это нельзя назвать объектно-ориентированным программированием, поскольку мы **type** не переопределяем, а вызываем напрямую. Давайте изменим это:

```
class UpperAttrMetaclass(type):

    def __new__(upperattr_metaclass, future_class_name,
                future_class_parents, future_class_attr):

        uppercase_attr = {}
        for name, val in future_class_attr.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

        # Повторное использование метода type.__new__
        # Никакой магии, это базовое ООП
        return type.__new__(upperattr_metaclass, future_class_name,
                            future_class_parents, uppercase_attr)
```

Наверное, вы заметили аргумент upperattr_metaclass. Этот метод первым аргументом получает текущий экземпляр. Точно так же, как и *self* для обычных методов. Имена аргументов такие длинные для наглядности, но для *self* все имена имеют названия обычной длины. Поэтому реальный метакласс будет выглядеть так:

```
class UpperAttrMetaclass(type):
```

```
    def __new__(cls, clsname, bases, dct):

        uppercase_attr = {}
        for name, val in dct.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

     return type.__new__(cls, clsname, bases, uppercase_attr)
```

Используя метод *super*, можно сделать код более "чистым":

```
class UpperAttrMetaclass(type):

    def __new__(cls, clsname, bases, dct):

        uppercase_attr = {}
        for name, val in dct.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

        return super(UpperAttrMetaclass, cls).__new__(cls,
clsname, bases, uppercase_attr)
```

Собственно, это все, что вам нужно знать о метаклассах. Причина сложности кода, который использует метаклассы, даже не в самих метаклассах. Код становится сложным, поскольку обычно метаклассы используют для сложных задач, основанных на наследовании и манипуляции такими переменными, как __dict__. Также посредством метаклассов вы можете:

- перехватить создание класса
- изменить класс
- вернуть измененный класс

20.3.3. ИСПОЛЬЗОВАНИЕ МЕТАКЛАССОВ ВМЕСТО ФУНКЦИЙ

Спрашивается, зачем использовать классы метаклассов вместо функций? А тому есть несколько причин:

- Более понятные идентификаторы. Например, когда вы читаете UpperAttrMetaclass(type), вы понимаете, что будет дальше.
- Можно использовать ООП. Метакласс может наследоваться от метакласса, переопределять родительские методы.
- Можно лучше структурировать свой код. Вряд ли вы будете использовать метаклассы для чего-то простого. Обычно это более сложные задачи. Возможность создавать несколько методов и группировать их в одном классе очень полезна, чтобы сделать код более удобным для чтения.
- Можете использовать __new__, __init__ и __call__. Это открывает простор для творчества. Обычно все это можно сделать в __new__, но некоторым программистам просто удобнее использовать __init__.

20.4. Генерация кода

Как уже было отмечено в начале этой главы, динамическая генерация кода – самый сложный способ метапрограммирования. Python предоставляет инструменты, позволяющие создавать и выполнять код, а также вносить изменения в уже откомпилированные части кода. Все это открывает практически безграничные возможности метапрограммирования. К сожалению, все это настолько сложные материи, что данному подходу можно посвятить отдельную книгу. Можете считать данный раздел как указатель направления. Мы укажем путь, по которому вы сможете дальше развиваться самостоятельно, если это вам нужно.

Python содержит три встроенные функции, позволяющие вручную выполнить, вычислить и откомпилировать произвольный код Python – *exec*, *eval*, *compile*.

Сигнатура функции *exec()* выглядит так:

```
exec(object, global, locals)
```

Данная функция позволяет выполнить код Python. Элемент **object** должен быть объектом кода (см. функцию *compile*) или же строкой, представляющим один оператор или последовательность нескольких. Аргументы *global* и *local* – это глобальные и локальные пространства имен для исполняемого кода, которые являются необязательными. Если они не указаны, то код будет выполнен в текущем пространстве. Если указаны, то *global* должен

быть словарем, а *local* может быть любым объектом отображения, он всегда возвращает *None*.

Сигнатура функции *eval* выглядит так:

```
eval(expression, global, locals)
```

Данная функция используется для вычисления выражения и возвращения его значения. Она похожа на exec(), но *expression* – это одно выражение, а не большой кусок кода. Функция возвращает значение вычисленного выражения.

Сигнатура функции *compile* такая:

```
compile(source, filename, mode)
```

Компилирует источник в объект кода или AST. Исходный код предоставляется в качестве строкового значения в аргументе *source*. Здесь *filename* — это файл, из которого читается код. Если связанного файла нет (например, если он был создан динамически), обычно используется значение <string>. Режим — *exec* (последовательность операторов), *eval* (одно выражение) или *single* (один интерактивный оператор, например, в интерактивной сессии Python).

Легче всего начать работу с функциями exec() и eval(), поскольку функции работают со строками. Если вы уже программировали на Python, то наверняка сможете сформировать рабочий исходный код из программы.

Наиболее полезная функция в контексте метапрограммирования – это функция exec(), поскольку она позволяет выполнить любую последовательность операторов Python. Однако при работе с этой функцией нужно быть осторожным. Вас должно беспокоить словосочетание "любую последовательность". Не нужно бояться сбоя Python – это не самое страшное. Самое страшное – это возможные проблемы с безопасностью вашего приложения, что может вам очень дорого стоить. Другими словами, функции exec() и eval() могут сделать ваше приложение уязвимым, поэтому нужно с особой осторожностью относиться к тому, что вы передаете на вход этим функциями. Одно дело, если код генерируете вы, другое дело, если его вводит пользователь или он поступает откуда-то извне (база данных, внешний файл и т.д.).

Даже если вы 100% доверяете входным данным (например, пишете приложение исключительно для себя, и входные данные будете формировать исключительно вы), вы должны понимать, что результаты работы вашей программы могут быть весьма неожиданными.

Первое, с чем вам придется столкнуться – это производительность. Перед тем, как поговорить о ней, давайте разберемся, что делает Python, если вы импортируете модуль (*import foo*):

1. Он выполняет поиск модуля. Это происходит при просмотре информации из sys.path разными способами. Есть встроенная логика импорта, есть ловушки импорта и, в целом, в этот процесс задействовано довольно много магии, о которой мы сейчас не будем говорить.
2. После того, как модуль найден, Python, в зависимости от того, найден откомпилированный код (.pyc) или исходный (.py), он производит некоторую работу. Если доступен байт-код и контрольная сумма соответствует текущей версии интерпретатора, временная метка файла байт-кода новее или равна исходной версии, он его загружает. Если байт-код отсутствует (есть только .py-файл) или же устарел, он загрузит исходный файл и откомпилирует его в байт-код. Для этого он проверяет магические комментарии в заголовке файла на предмет настроек кодирования и декодирует в соответствии с этими настройками. Он также проверит, существует ли специальный комментарий ширины табуляции для обработки табуляции как чего-то другого, кроме 8 символов, если это необходимо. Некоторые хуки импорта затем будут генерировать файлы .pyc или сохранять байт-код в другом месте (__pycache__) в зависимости от версии и реализации Python.
3. Интерпретатор Python создает новый объект модуля (вы можете сделать это самостоятельно, вызвав imp.new_module или создав экземпляр types.ModuleType – это эквивалентно) с собственным именем.
4. Если модуль был загружен из файла, устанавливается ключ __file__. Система импорта также будет следить за тем, чтобы __package__ и __path__ были установлены правильно, если пакеты задействованы до выполнения кода. Хуки импорта также устанавливают переменную __loader__.
5. Интерпретатор Python выполняет байт-код в контексте словаря модуля. Таким образом, локальные и глобальные фреймы для исполняемого кода являются атрибутом __dict__ этого модуля.
6. Модуль вставляется в sys.modules.

Ни один из вышеперечисленных шагов никогда не передавал строку ключевому слову или функции *exec*. Это, очевидно, правда, потому что все эти действия происходят глубоко внутри интерпретатора Python, если вы не используете ловушку импорта, написанную на Python. Но даже если интерпретатор Python был написан на Python, он никогда не передал бы строку в функцию *exec*. Итак, что бы вы хотели сделать, если хотите сами преобразовать эту строку в байт-код? Вы бы использовали встроенную компиляцию:

```
>>> code = compile('a = 1 + 2', '<string>', 'exec')
>>> exec code
>>> print a
3
```

Как видите, *exec* тоже успешно выполняет байт-код (не обязательно передавать строку). Поскольку переменная кода на самом деле является объектом типа *code*, а не строкой. Второй аргумент для компиляции – подсказка имени файла. Если мы компилируем из реальной строки, мы должны указать значение, заключенное в угловые скобки, потому что это то, что будет делать Python. <string> и <stdin> – общие значения. Если у вас есть файл, укажите здесь фактическое имя файла. Последний параметр может быть одним из "exec", "eval" или "single". Первый – это то, что использует *exec*, второй – то, что использует функция *eval*. Разница в том, что первый может содержать операторы, второй – только выражения. 'single' – это разновидность гибридного режима, который бесполезен ни для чего, кроме интерактивных оболочек. Он существует исключительно для реализации таких вещей, как интерактивная оболочка Python, и очень ограничен в использовании.

Однако здесь мы уже использовали функцию, которую вы никогда и никогда не должны использовать: выполнение кода в пространстве имен вызывающего кода. Что делать вместо этого? Выполнить в новой среде:

```
>>> code = compile('a = 1 + 2', '<string>', 'exec')
>>> ns = {}
>>> exec code in ns
>>> print ns['a']
3
```

Зачем так делать? Более понятное средство для начинающих, поскольку *exec* без словаря должен обойти некоторые детали реализации в интерпретаторе. Мы поговорим об этом позже.

> ***Примечание***. На данный момент: если вы хотите использовать exec и планируете выполнять этот код более одного раза, убедитесь, что вы сначала скомпилировали его в байт-код, а затем выполняете только этот байт-код, и только в новом словаре в качестве пространства имен.

Однако в Python 3 оператор *exec ... in* исчез, и вместо этого вы можете использовать новую функцию *exec*, которая принимает глобальные и локальные словари в качестве параметров.

А вот теперь мы можем поговорить о производительности. Насколько быстрее выполняется байт-код по сравнению с созданием байт-кода и его выполнением:

```
$ python -mtimeit -s 'code = "a = 2; b = 3; c = a * b"' 'exec
code'
10000 loops, best of 3: 22.7 usec per loop

$ python -mtimeit -s 'code = compile("a = 2; b = 3; c = a *
b",
  "<string>", "exec")' 'exec code'
1000000 loops, best of 3: 0.765 usec per loop
```

В 32 (!) раза быстрее даже на таком простом примере. И чем больше у вас кода, тем хуже становится ситуация. Почему так происходит? Поскольку синтаксический анализ кода Python и преобразование его в байт-код – дорогостоящая операция по сравнению с оценкой байт-кода.

Хорошо, урок усвоен. Сначала компиляция, а затем выполнение уже откомпилированного кода. Но что еще нужно учитывать при использовании *exec*? Следующее, что вы должны помнить, это огромная разница между глобальной и локальной областью видимости. Хотя и глобальная, и локальная область видимости используют словари в качестве хранилища данных, последняя на самом деле нет. Локальные переменные в Python просто берутся из локального словаря фрейма и помещаются туда по мере необхо-

димости. Для всех вычислений, которые происходят между ними, словарь никогда не используется. Вы можете быстро убедиться в этом сами.

Выполните в интерпретаторе Python следующее:

```
>>> a = 42
>>> locals()['a'] = 23
>>> a
23
```

Работает как положено. Почему? Потому что интерактивная оболочка Python выполняет код как часть глобального пространства имен, как и любой код вне функций или объявлений классов. Локальная область видимости – это глобальная область:

```
>>> globals() is locals()
True
```

Что произойдет, если мы попытаемся сделать то же самое, но на уровне функции:

```
>>> def foo():
...  a = 42
...  locals()['a'] = 23
...  return a
...
>>> foo()
42
```

Совсем не то, что мы ожидали, правда? Но это еще раз красочно демонстрирует, что локальные переменные – это не совсем локальные, по крайней мере, в контексте программы, а не функции. Об этом нужно помнить просто при программировании, не говоря уже об использовании exec/eval.

А что можно сказать о производительности кода, выполняемого в глобальной области по сравнению с кодом, который выполняется в локальной области? Это намного сложнее измерить, потому что модуль *timeit* по умолчанию не позволяет нам выполнять код в глобальной области видимости. Поэтому нам нужно будет написать небольшой вспомогательный модуль, который эмулирует это:

```
code_global = compile('''
sum = 0
for x in xrange(500000):
    sum += x
''', '<string>', 'exec')
code_local = compile('''
def f():
    sum = 0
    for x in xrange(500000):
        sum += x
''', '<string>', 'exec')

def test_global():
    exec code_global in {}

def test_local():
    ns = {}
    exec code_local in ns
    ns['f']()
```

Здесь мы дважды компилируем один и тот же алгоритм в строку. Один раз напрямую глобально, один раз – завернутый в функцию. Получается, что у нас есть две функции. Первая выполняет этот код в пустом словаре, вторая выполняет код в новом словаре, а затем вызывает объявленную функцию. А теперь мы можем использовать *timeit* для вычисления нашей скорости:

```
$ python -mtimeit -s 'from execcompile import test_global as t' 't()'
10 loops, best of 3: 67.7 msec per loop

$ python -mtimeit -s 'from execcompile import test_local as t' 't()'
100 loops, best of 3: 23.3 msec per loop
```

Снова мы получили прирост производительности, правда, не такой ощутимый, как в прошлый раз. Однако, мы используем всего 100 циклов, при этом прирост составил почти 200%, то есть мы стали в три раза быстрее, и один цикл у нас выполняется 23.3 секунды против 67.7 секунд.

Почему так получается? Это связано с тем, что быстрые локальные переменные быстрее словарей. В локальной области видимости Python отслеживает имена переменных, о которых он знает. Каждой из этих переменных

присваивается номер (индекс). Этот индекс используется в массиве объектов Python вместо словаря. Он вернется к словарю только в том случае, если это необходимо (в целях отладки, в случае использования *exec* и т.д.). Несмотря на то, что *exec* все еще существует в Python 3 (как функция), вы больше не можете переопределять переменные в локальной области. Компилятор Python не проверяет, используется ли встроенная функция *exec*, и из-за этого не будет неоптимизировать область видимости.

Все вышеперечисленные знания полезно знать, если вы планируете использовать интерпретатор Python для интерпретации вашего собственного языка путем генерации кода Python и компиляции его в байт-код. Так, например, работают механизмы шаблонов, такие как Mako, Jinja2 или Genshi.

Однако, большинство людей используют оператор *exec* для выполнения реального кода Python из разных мест. Очень популярный кейс – выполнение файлов конфигурации как кода Python. Так, например, делает Flask. Обычно это вполне нормально, потому что вы не ожидаете, что ваш файл конфигурации будет местом, будет реализован реальный код. Однако есть люди, которые используют *exec* для загрузки реального кода Python, объявляющего функции и классы. Это очень популярный подход в некоторых системах плагинов и фреймворке web2py.

Почему это не очень хорошая идея? Да потому что она ломает некоторые негласные соглашения относительно кода Python, а именно:

1. Классы и функции принадлежат модулю. Это основное правило справедливо для всех функций и классов, импортированных из обычных модулей:

```
>>> from xml.sax.saxutils import quoteattr
>>> quoteattr.__module__
'xml.sax.saxutils'
```

Почему это важно? Посмотрим, как работает Pickle:

```
>>> pickle.loads(pickle.dumps(quoteattr))
<function quoteattr at 0x1005349b0>
>>> quoteattr.__module__ = 'fake'
>>> pickle.loads(pickle.dumps(quoteattr))
Traceback (most recent call last):
  ..
```

```
pickle.PicklingError: Can't pickle quoteattr: it's not found
as fake.quoteattr
```

Если вы используете exec для выполнения кода Python, будьте готовы к

тому, что некоторые модули, такие как pickle, inspect, pkgutil, pydoc и, возможно, некоторые другие, которые зависят от них, не будут работать должным образом.

2. В Python встроен **циклический сборщик мусора**, классы могут иметь деструкторы, а завершение работы интерпретатора приводит к прерыванию циклов. Что это значит? CPython внутренне использует рефсчет. Один из многих недостатков подсчета ссылок заключается в том, что он не может обнаруживать циклические зависимости между объектами. Таким образом, в какой-то момент Python представил циклический сборщик мусора.

Однако Python также позволяет использовать деструкторы для объектов. Однако деструкторы означают, что циклический сборщик мусора пропустит эти объекты, потому что он не знает, в каком порядке он должен удалить эти объекты.

Теперь давайте посмотрим на невинный пример:

```
class Foo(object):
    def __del__(self):
        print 'Deleted'
foo = Foo()
```

Давайте выполним этот файл:

```
$ python test.py
Deleted
```

Выглядит нормально. Теперь выполним его же через exec:

```
>>> execfile('test.py', {})
>>> execfile('test.py', {})
>>> execfile('test.py', {})
>>> import gc
>>> gc.collect()
27
```

Он что-то почистил, но он никогда бы не очистил наш экземпляр Foo. Что же происходит? Происходит неявный цикл между *foo* и самой функцией __del__. Функция знает область видимости, в которой она была создана, и из экземпляра __del__ -> global scope -> foo, у нее есть хороший цикл.

Теперь, когда мы знаем причину, почему этого не происходит, если у вас есть модуль? Причина в том, что Python выполняет некоторый трюк при закрытии модуля. Он переопределит все глобальные значения, которые не начинаются с подчеркивания с *None*. Мы можем легко убедиться в этом, если введем значение *foo* вместо *Deleted*:

```
class Foo(object):
    def __del__(self):
        print foo
foo = Foo()
```

Что получим в итоге? Конечно, *None*:

```
$ python test.py
None
```

Поэтому при использовании exec() и компании нужно быть предельно осторожным, поскольку это может стать причиной утечек памяти.

3. Срок службы объектов. Глобальное пространство имен сохраняется с момента его импорта до момента завершения работы интерпретатора. С *exec* вы, как пользователь, больше не знаете, когда это произойдет. Это могло случиться раньше в случайном месте. web2py здесь является частым нарушителем. В web2py волшебно исполняемое пространство имен приходит и уходит с каждым запросом, что является очень неожиданным поведением для любого опытного разработчика Python.

Наконец, в завершении этой главы поговорим о PHP и Python. Многие опытные веб-разработчики знакомы с обоими этими языками. Помните, что Python – это не PHP. Не пытайтесь обойти идиомы Python, потому что какой-то другой язык (наш любимый PHP) делает это иначе. Пространства имен находятся в Python по какой-то причине, и то, что он дает вам инструмент *exec*, не означает, что вы должны использовать этот инструмент. Язык С дает вам *setjmp* и *longjmp*, но вы будете очень осторожны при их использовании. Комбинация *exec* и *compile* – мощный инструмент для всех, кто хочет реализовать специфичный для предметной области язык поверх

Python, или для разработчиков, заинтересованных в расширении (а не обходе) системы импорта Python.

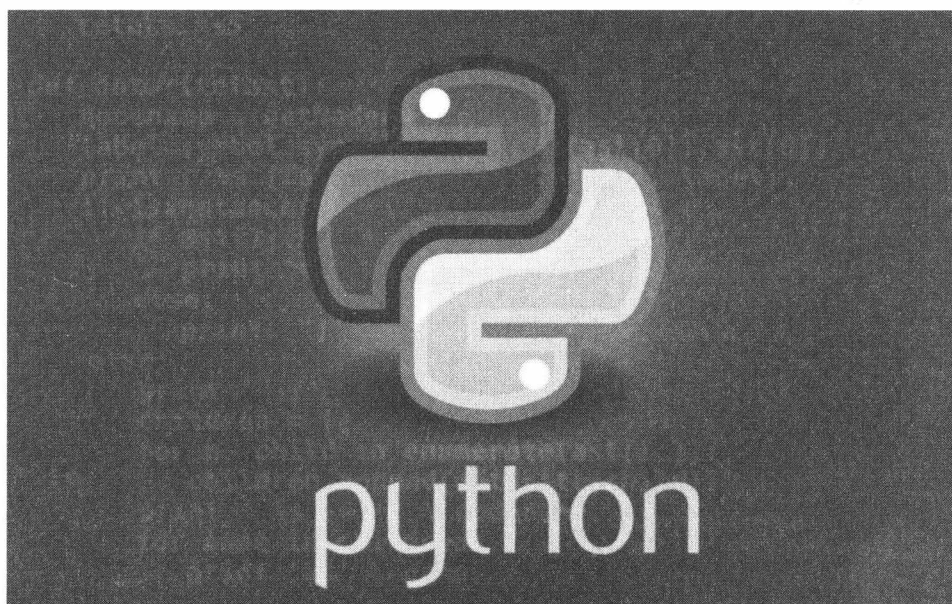
Однако web2py и его использование *execfile()* – не единственные нарушители в веб-сообществе Python. Werkzeug также изрядно злоупотребляет соглашениями Python. Система импорта по требованию вызывает больше проблем, чем решает, и в настоящее время разработчики Python от нее отказываются (несмотря на все свое нежелание этого делать).

Django также злоупотреблял внутренними компонентами Python. Она генерировала код Python на лету и полностью меняла семантику (до такой степени, что импорт исчезал без предупреждения!). Разработчики Django тоже усвоили урок и исправили эту проблему. То же самое касается и web.py, который злоупотреблял оператором *print* для записи во внутренний локальный буфер потока, который затем был отправлен в качестве ответа браузеру. Также кое-что, что оказалось плохой идеей, было впоследствии удалено.

Несмотря на все возможности, открываемые функцией *exec()*, мы призываем отказаться от ее использования в пользу обычных модулей Python. Если разработчик Python начнет свое путешествие в запутанном мире неправильно выполненных модулей Python, он будет, как минимум, сбит с толку, когда продолжит свое путешествие в другой среде Python. Наличие разной семантики в разных фреймворках/модулях/библиотеках очень вредно для Python как среды выполнения, и как языка.

ГЛАВА 21.

КОНТРОЛЬ КОДА



В современном мире невозможно обойтись без системы управления версиями (она же система контроля версий). Программные продукты становятся все сложнее и сложнее, над ними работает целая команда и управлять таким проектом без системы контроля версий практически невозможно. Даже если вы – программист-одиночка, система управления версиями так же будет полезна, ведь порой очень сложно вспомнить, какие изменения вносились в код неделю назад, не говоря уже про более длительные периоды. А в случае с командной работой система контроля кода подскажет не только, что и когда изменялось, но и кто вносил те или иные изменения.

21.1. Введение в системы контроля версиями

Системы управления версиями (*Version Control Systems*, *VCS*) предоставляют возможность общей работы над любыми файлами, но все их преимущества можно получить только при работе с текстовыми файлами. И при этом совершенно не важно, на каком языке вы программируете, вообще не

важно, программируете ли вы или пишете роман. Главное, чтобы основным типом обрабатываемой информации был текст.

Системы контроля версиями (СКВ) бывают централизованными и распределенными. Централизованная СКВ представляет собой один сервер с файлами, позволяющий пользователям вносить свои и видеть чужие изменения в контролируемый набор файлов, называемый проектом. Принцип очень и очень прост – каждый может скопировать нужные файлы на свой компьютер, внести в них изменения, а после – загрузить изменения на сервер. После применения изменений генерируется номер версии. Другие пользователи могут получить измененные файлы путем синхронизации их копий проекта через механизм обновления. Другими словами, если вы и Вася работаете над проектом, а Вася вечером внес изменения в какие-то файлы, то спустя некоторое время, например, утром, когда вы включите компьютер и приступите к работе, вы:

- Получите самую новую версию проекта.
- Увидите, кто и какие изменения вносил в предыдущую версию.
- Сможете сравнить версии файлов.

На вашем компьютере будет установлено специальное программное обеспечение, которое "видит", что проект изменен и загрузит измененные файлы на ваш компьютер. Аналогично, если вы внесете изменения в проект, они будут загружены на сервер и другие члены команды увидят, какие изменения вы вносили. Это очень удобно, а главное позволяет с точностью до последнего символа контролировать код.

Такой процесс прекрасно работает в проектах, над которыми трудятся несколько разработчиков и где относительно небольшое количество файлов. Но все становится сложнее, когда вырастает количество разработчиков и размер кода. Сложные изменения часто затрагивают множество файлов, разные разработчики практически в одно и то же время вносят изменения, что порождает цепочку не совсем логичных версий, а проект разрастается до таких размеров, что некоторые программисты просто не в состоянии хранить у себя локальную копию всего проекта.

Назревают две проблемы:

1. Программист может долго работать только в своей локальной копии без резервного копирования.

2. Сложно делиться своей работой с другими, пока она не отправлена в хранилище, а отправлять ее без проверки и тестирования означает поставить под угрозу работу всего проекта.

В централизованной СКВ подобные проблемы решаются посредством ответвлений и слияний. От основного потока изменения могут ответвляться ветви (*форки*), которые снова сливаются в основной поток. Но в этом случае возникает другая проблема. Представьте себе, что есть основной поток, над которым работает множество программистов. Программист Вася создает свой форк и вносит в него изменения, работает над проектом неделю или даже больше и затем хочет выполнить слияние своей ветки с основным проектом. В процессе слияния выясняется, что пользователь Марк также вносил изменения в файлы, которые правил Василий – уже после того, как Василий создал свой форк. Чьи изменения считать правильными – Марка или Василия?

Вот в этом и заключаются основные недостатки централизованной СКВ:

1. Ветвление и слияния сложно организовать. В сложных проектах такая организация управления кодом может превратиться в настоящий ночной кошмар.
2. Поскольку система централизована, нельзя зафиксировать изменения в автономном режиме. То есть вы не можете зафиксировать результаты своей работы за вчера, чтобы сравнить их с сегодняшними. Вам нужно только отправлять изменения на сервер, чтобы выполнить централизованный *commit*, что в свою очередь породит создание новой версии. Когда над проектом работает с десяток программистов (а это немного) и каждый будет ежедневно делать коммит, то накопиться очень много версий, в которых очень сложно будет разобраться.
3. Такая схема практически не работает для проектов, где у компаний есть собственный филиал программного обеспечения и нет центрального хранилища, в котором бы у каждого была своя учетная запись.

Внимание! Еще нужно помнить о следующем недостатке централизованной СКВ. У каждого клиента есть копия всего исходного кода и внесенных изменений. В этом случае, если один из серверов выйдет из строя, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы.

Несмотря на эти недостатки, централизованные СКВ все еще популярны в небольших проектах и даже в крупных компаниях за счет инертности корпоративной среды. Другими словами, когда-то, лет 10 назад, решили, что будут использовать систему А и все эти 10 лет она используется, потому что так принято.

В качестве примера централизованных СКВ можно привести Subversion (SVN) и System Concurrent Version (CVS).

Многие крупные проекты из-за недостатков централизованных СКВ перешли на распределенную архитектуру. Распределенные СКВ решают недостатки централизованных коллег. Они размещаются не на основном сервере, с которым работают программисты, а на равноправных (*peer-to-peer*) узлах. У каждого программиста есть свой независимый репозиторий для проекта, и он синхронизирует его с другими репозиториями.

Здесь больше свободы. Например, Марк может загрузить файлы из какого-то основного хранилища (пусть оно называется PRJ). Марк вносит изменения в некоторые файлы. Василий загружает файлы из хранилища Марка и тоже вносит изменения. Затем Василий отправляет изменения в PRJ. Денис загружает файлы из PRJ и вносит в них изменения, а затем загружает их в PRJ.

Другими словами, здесь есть какой-то один центральный репозиторий, но вам не обязательно с ним работать. Вы можете использовать репозитории других программистов, в зависимости от того, как организована работа людей и управление проектом. Здесь больше нет какого-то центрального сервера, на который пользователи вносят изменения, а менеджер проекта определяет стратегию управления кодом – загрузки, выгрузки, внесения изменений.

Недостаток распределенной системы в том, что программисты вынуждены больше думать. В распределенной системе нет больше какого-то глобального номера, с которым можно сверяться. Таким образом, нужно задействовать дополнительные текстовые метки (теги), которые могут быть присоединены к версии.

При работе с распределенной системой получается, что каждый может вносить какие-либо изменения, загруженные у кого-либо. Может получиться бардак. Поэтому при работе с распределенной системой также создают центральный сервер, но его назначение совсем иное, чем у централизованных СКВ. Этот сервер представляет собой хаб, позволяющий всем программистам одновременно использовать свои изменения в одном месте, а не заниматься загрузкой и выгрузкой друг у друга. Такой репозиторий часто назы-

вается вышестоящим и используется также в качестве резервного для всех изменений, выполняемых в отдельных репозиториях всех членов команды.

Для объединения доступа к коду в распределенной СКВ используются различные подходы. Самый простой заключается в создании сервера, который будет выполнять функцию обычного централизованного сервера, и каждый программист может вносить изменения в общий поток. Но такой подход не позволит вам получить все преимущества распределенной СКВ, поскольку работа будет организована так же, как и в случае с централизованной СКВ, а поменяется просто название системы СКВ. Будете использовать Git, гордо заявлять об этом, а работа будет построена так же, как в случае с SVN. Для небольших проектов такой вариант использования распределенной СКВ допускается – чтобы программисты сразу привыкали к использованию нормальной системы контроля версий. В сложных проектах принято использовать другой подход, при котором на сервере создаются несколько репозиториев с разными уровнями доступа:

- *Нестабильный репозиторий* – в него может вносить изменения каждый программист.
- *Стабильный репозиторий* – с него может выгружать файлы любой программист, а вносить изменения могут только менеджеры проекта – они решают, что нужно изменять, какие изменения нужно вносить из нестабильного репозитория.
- *Релизный репозиторий* – используется для релизов, доступен только для чтения.

При таком подходе каждый участник может вносить изменения, а менеджеры могут решать, чьи изменения включить в стабильный репозиторий, а какие – нет.

Какую систему выбрать – централизованную или распределенную? Сейчас мы сэкономим вам кучу времени, и вы перестанете ломать голову над этим вопросом. Забудьте о централизованных системах – это пережиток прошлого и если вы только выбираете, какую систему СКВ использовать, ни в коем случае не выбирайте централизованные системы. Только распределенная.

Хорошо, следующий вопрос – какую распределенную систему выбрать? Здесь тоже все просто – используйте Git. Git – это самая популярная распределенная система управления версиями, знаменитая тем, что была создана Линусом Торвальдсом (создатель Linux) для работы над кодом ядра

Linux, когда потребовалось отказаться от патентованного ПО BitKeeper, которое использовалось ранее для работы над кодом ядра.

Git – лучшая, но не идеальная система. В том, что она лучшая – никто не сомневается, ведь она используется для управления кодом ядра Linux – это очень сложный проект, содержащий тонны кода (когда количество строк кода в ядре Linux превысило 15 миллионов, эти строки перестали считать!). Но Git также имеет свои недостатки. В первую очередь – это довольно сложная система, особенно для новичков. В последнее время эта проблема частично решается наличием специального ПО, облегчающего работу с Git, например, GitHub Desktop (рис. 21.3). Приложение позволяет создавать копию репозитория, выгружать изменения в репозиторий, просматривать эти изменения и т.д. Все же удобнее, чем командная строка.

Да, вы будете привязаны к сервису GitHub. GutHub – это крупнейший веб-сервис для хостинга ИТ-проектов и их совместной разработки, основан на системе Git. С одной стороны, привязка к GitHub, с другой – этот хостинг бесплатный, за него не нужно платить, не нужно разворачивать отдельный сервер для Git, можно создать репозиторий, скачать GitHub Desktop и приступить к работе.

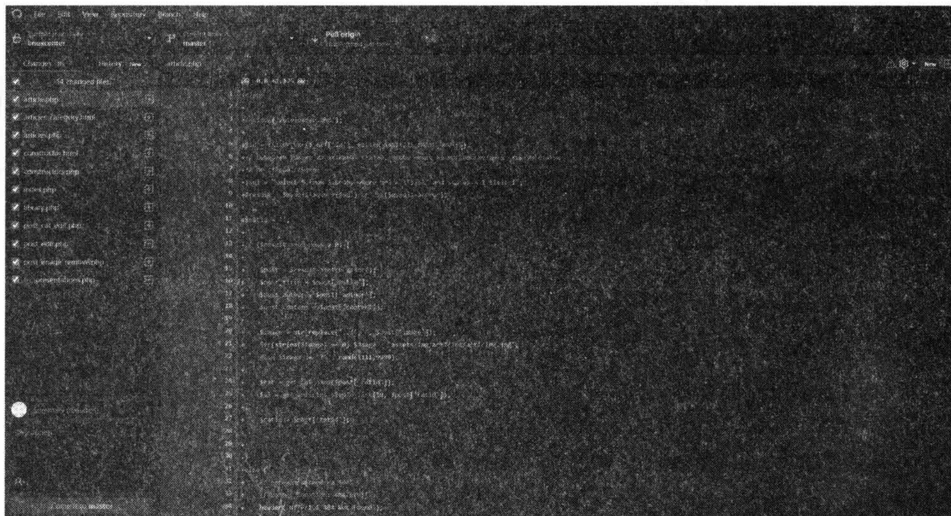


Рис. 21.1. Приложение GitHub Desktop

Здесь важно понимать разницу между Git и GitHub. GitHub — сервис онлайн-хостинга репозиториев, обладающий всеми функциями распределенного контроля версий и функциональностью управления исходным кодом — все, что поддерживает Git и даже больше. Также GitHub может похвастаться контролем доступа, багтрекингом, управлением задачами и вики для каждого проекта.

Git-репозиторий, загруженный на GitHub, доступен с помощью интерфейса командной строки Git и Git-команд. Также есть и другие функции: документация, запросы на принятие изменений (*pull requests*), история коммитов, интеграция со множеством популярных сервисов, email-уведомления, эмодзи, графики, вложенные списки задач, система @упоминаний, похожая на ту, что в Twitter, и т.д.

Другими словами, Git – это инструмент, а GitHub – это сервис, основанный на этом инструменте. Git – это как собственный автомобиль, а GitHub – это как автомобиль напрокат, только в отличие от прокатного автосервиса, он является бесплатным.

21.2. Знакомство с Git

Git – это распределенная система контроля версий, созданная, как уже отмечалось, создателем ядра Linux Линусом Торвальдсом для работы над исходным кодом ядра. До появления Git для разработки ядра Linux использовался коммерческий продукт – BitKeeper VCS, который обеспечивал сложные операции, недоступные в бесплатных системах контроля версий того времени, таких как RCS и CVS (*Concurrent Version System*). Однако компания, владеющая BitKeeper, установила дополнительные ограничения на его "свободную" версию, выпущенную в 2005 году, и сообщество Linux поняло, что BitKeeper больше не может использоваться.

Тогда Линус начал искать альтернативные решения. Сторонясь коммерческих решений, он изучил пакеты бесплатного программного обеспечения, но нашел те же ограничения и недостатки, которые были и раньше. Что же не устраивало его в существующих СКВ? Каковы были неуловимые недостающие возможности или характеристики, которые Линус хотел и не мог найти?

РАСПРЕДЕЛЕННАЯ РАЗРАБОТКА

Есть много граней распределенной разработки и Линус хотел новую VCS, которая бы покроет большинство из них. Она должна поддерживать парал-

лельную, а также независимую и одновременную разработку в частных репозиториях без потребности в постоянной синхронизации с центральным репозиторием, который мог бы сформировать узкое место разработки. В результате многие разработчики, находящиеся в разных местах, могли бы работать над проектом, даже если некоторые из них временно были оффлайн.

МАСШТАБИРУЕМОСТЬ, СПОСОБНАЯ ПОДДЕРЖИВАТЬ ТЫСЯЧИ РАЗРАБОТЧИКОВ

Недостаточно иметь только распределенную модель разработки. Линус знал, что над выпуском каждой версии Linux трудятся тысячи разработчиков. Таким образом, любая новая СКВ должен был поддерживать очень большое количество разработчиков, независимо от того, работают они над одними и теми же или разными частями проекта. Новая СКВ должна быть в состоянии надежно интегрировать всю их работу.

БЫСТРАЯ И ЭФФЕКТИВНАЯ РАБОТА

Нужно было гарантировать, что новая СКВ быстра и эффективна. Отдельные операции обновления и операции передачи по сети должны быть очень быстрыми. А чтобы сэкономить дисковое пространство и сократить время передачи данных по сети, нужно использовать сжатие и методы "дельты". Использование распределенной модели вместо централизованной также гарантирует, что сетевая задержка не будет препятствовать ежедневной разработке.

ПОДДЕРЖКА ЦЕЛОСТНОСТИ И ДОВЕРИЕ

Поскольку Git – это распределенная система управления версиями, жизненно важно получить уверенность в том, что обеспечивается целостность данных, и никто не может внести несанкционированные изменения. Как вы узнаете, что данные не были изменены при переходе от одного разработчика к другому? Или от одного репозитория к следующему? Git использует общую криптографическую хеш-функцию, названную Secure Hash Function (SHA1), для идентификации объектов в базе данных. Данная функция гарантирует целостность и доверие для распределенных репозиториев Git.

ОТСЛЕЖИВАЕМОСТЬ

Один из ключевых аспектов системы управления позволяет узнать, кто изменил файлы, и, если это возможно, почему. Git ведет журнал изменений на каждой фиксации, изменяющей файл. Какая именно информация будет храниться в журнале, определяется до начала проекта в зависимости от требования к проекту, управлению, разных соглашений и т.д. Git гарантирует, что в ваших файлах не будет загадочных изменений, поскольку он отслеживает каждое изменение.

НЕИЗМЕННОСТЬ

База данных репозитория Git содержит объекты данных, которые являются неизменными. После их создания и помещения в базу данных они больше не могут быть изменены. Они могут быть воссозданы иначе, конечно, но исходные данные не могут быть изменены без последствий. Проект базы данных Git означает, что вся история, сохраненная в базе данных управления версиями, также неизменная. Использование неизменных объектов имеет несколько преимуществ, в том числе быстрое сравнение для равенства.

АТОМАРНЫЕ ТРАНЗАКЦИИ

При использовании атомарных транзакций много разных, но связанных изменений, выполняются вместе или не выполняются вообще. Это гарантирует, что базу данных управления версиями не оставят в частично измененном или поврежденном состоянии при обновлении или фиксации. Git реализует атомарные транзакции, записывая полные, дискретные состояния репозитория, которые не могут быть разделены на отдельные или меньшие изменения состояния.

ПОДДЕРЖКА И ПООЩРЕНИЕ РАЗВЕТВЛЕНИЯ РАЗРАБОТКИ

Почти все СКВ могут поддерживать различные генеалогии разработки в единственном проекте. Например, одну последовательность изменений

кода можно было вызвать "разработкой", а другую – "тестированием". Каждая система управления версиями может также разделить одну линию разработки на несколько линий, а затем объединить их в одно целое. Git называет каждую линию разработки ветвью и назначает каждой ветви собственное имя.

Вместе с ветвлением приходит объединение. Линус хотел не только возможность ветвления, но и простое объединение те ответвлений. Поскольку слияние ответвлений часто было болезненной и трудной работой в системах управления версиями, важно поддерживать быстрое и простое слияние.

ПОЛНЫЕ РЕПОЗИТОРИИ

Чтобы отдельные разработчики не запрашивали журнал изменений у централизованного сервера репозитория, важно, чтобы у каждого репозитория была полная копия всех изменений каждого файла.

21.3. Установка Git

Для установки Git в Linux нужно ввести команду:

```
sudo apt install git git-doc gitweb git-gui gitk git-email
git-svn
```

Данная команда установит Git в Debian-подобном дистрибутиве (Ubuntu, Kubuntu, Debian и т.д.). Если у вас Fedora, то команду *apt* нужно заменить на *dnf*:

```
sudo dnf install git git-doc gitweb git-gui gitk git-email
git-svn
```

Git для Windows можно получить по адресу:

https://gitforwindows.org/

и устанавливается он как обычное Windows-приложение. Подробно описывать процесс установки нет смысла, так как любой среднестатистический Python-программист справится с этой задачей.

21.4. Основы работы с Git

Git управляет изменениями. Git – одна из систем управления версиями. Множество принципов, например, понятие фиксации, журнала изменений, репозитария, являются одинаковыми для всех инструментов подобного рода. Но Git предлагает много новинок. Некоторые понятия и методы других систем управления версиями могут работать по-другому в Git или не работать вообще. Независимо от наличия вашего опыта работы с подобными системами, данная книга учит вас работать с Git.

21.4.1. КОМАНДНАЯ СТРОКА GIT

Git очень просто использовать. Просто введите *git* без всяких аргументов. Git выведет свои параметры и наиболее общие подкоманды.

```
$ git

git [--version] [--exec-path[=GIT_EXEC_PATH]]
[-p|--paginate|--no-pager] [--bare] [--git-dir=GIT_DIR]
[--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]
```

Далее приведены наиболее часто используемые команды *git*:

- *add* – Добавляет содержимое файла в индекс
- *bisect* – Выполняет бинарный поиск ошибки по истории фиксаций
- *branch* – Выводит, создает или удаляет ветки
- *checkout* – Проверяет и переключает на ветку
- *clone* – Клонирует репозиторий в новый каталог
- *commit* – Записывает изменения в каталог (фиксация)

- *diff* – Показывает изменения между фиксациями, рабочими деревьями и т.д.
- *fetch* – Загружает объекты и ссылки из другого репозитория
- *grep* – Выводит строки, соответствующие шаблону
- *init* – Создает пустой репозиторий git или переинициализирует существующий
- *log* – Показывает журналы фиксации
- *merge* – Объединяет две или больше истории разработки
- *mv* – Перемещает или переименовывает файл, каталог или символическую ссылку
- *pull* – Получает изменения из удаленного репозитория и объединяет их с локальным репозиторием или локальной веткой
- *push* – Загружает изменения из вашего локального репозитория в удаленный
- *rebase* – Строит ровную линию фиксаций
- *reset* – Сбрасывает текущий HEAD в указанное состояние
- *rm* – Удаляет файлы из рабочего дерева и из индекса
- *show* – Показывает различные типы объектов
- *status* – Показывает состояние рабочего дерева
- *tag* – Создает, выводит, удаляет или проверяет тег объекта, подписанного с GPG

Чтобы вывести полный список подкоманд *git*, введите:

```
git help --all
```

Как видно из подсказки использования, имеется множество полезных опций для *git*. Большинство опций, показанных как [ARGS], применяются к определенным подкомандам.

Например, опция --version применяется к команде *git* и производит вывод номера версии:

```
$ git --version
git version 1.8.3.msysgt.0
```

А опция --amend, наоборот, относится к подкоманде *commit*:

```
$ git commit --amend
```

В некоторых случаях придется указывать обе формы опция, например:

```
$ git --git-dir=project.git repack -d
```

Получить справку по каждой подкоманде *git* можно следующими способами:

```
git help subcommand, git --help subcommand или git subcommand
--help
```

Раньше у Git был набор простых, отличных, автономных команд, разработанных согласно философии "Unix toolkit: небольшие и независимые инструменты. Каждая команда начиналась со строки *git* и содержала дефис, после которого следовало название выполнимого действия, например, *git-commit* и *git-log*. Однако на данный момент считается, что должна быть одна единственная исполнимая программа *git*, а выполняемые действия должны передаваться в качестве параметров, например, *git commit* и *git log*. Нужно отметить, что формы команд *git commit* и *git-commit* идентичны.

> ***Примечание.*** Онлайн-документация по Git доступна по адресу https://mirrors.edge.kernel.org/pub/software/scm/git/docs/

Команды *git* понимают "короткие" и "длинные" команды. Например, следующие две команды *git commit* эквиваленты:

```
$ git commit -m "Fixed a typo."
$ git commit --message="Fixed a typo."
```

В короткой форме используется один дефис, а длинная форма использует два. Некоторые опции существуют только в одной форме.

Наконец, вы можете разделить параметры из списка аргументов, используя "пустое двойное тире". Например, используйте двойное тире, чтобы отделить часть управления командной строки от списка операндов, например, от имен файлов:

```
$ git diff -w master origin -- tools/Makefile
```

Двойное тире нужно использовать, чтобы разделить и явно идентифицировать имена файлов, если иначе они могли бы быть приняты за другую часть команды. Например, если у вас есть тег "main.c" и файл "main.c", вы получите различное поведение:

```
# Проверка тега с именем "main.c"
$ git checkout main.c
# Проверка файла с именем "main.c"
$ git checkout -- main.c
```

21.4.2. БЫСТРОЕ ВВЕДЕНИЕ В GIT

Чтобы увидеть *git* в действии, давайте создадим новый репозиторий и добавим в него некоторое содержимое, а после чего сделаем несколько версий этого содержимого.

Существует две фундаментальных техники установки репозитория Git. Вы можете создать его с нуля, а потом заполнить его содержимым, или же скопировать, или как говорят в мире Git, склонировать, существующий репозиторий. Проще начать с пустого репозитория, поэтому давайте сделаем это.

СОЗДАНИЕ НАЧАЛЬНОГО РЕПОЗИТОРИЯ

Чтобы смоделировать типичную ситуацию, давайте создадим репозиторий для вашего персонального сайта в каталоге ~/public_html.

Если у вас нет вашего личного сайта в ~/public_html, создайте этот каталог и поместите в него файл index.html с любой строкой:

```
$ mkdir ~/public_html
$ cd ~/public_html
$ echo 'Мой сайт жив!' > index.html
```

Чтобы превратить ~/public_html или любой другой каталог в репозиторий Git, просто запустите *git init*:

```
$ git init
Initialized empty Git repository in .git/
```

Программе *git* все равно, начинаете ли вы с полностью пустого каталога или с каталога, полного файлов. Процесс преобразования каталога в репозиторий Git будет одинаковым.

Чтобы отметить, что ваш каталог является репозиторием Git, команда *git init* создает скрытый каталог с названием .git, находящийся на верхнем уровне вашего проекта.

Все остальное в каталоге ~/public_html останется нетронутым. Git считает этот каталог рабочим каталогом вашего проекта, в котором вы изменяете свои файлы. Git интересуют только файлы из скрытого каталога .git.

ДОБАВЛЕНИЕ ФАЙЛОВ В ВАШ РЕПОЗИТОРИЙ

Команда *git init* создает новый репозиторий Git. В самом начале каждый репозиторий Git будет пустым. Чтобы управлять контентом, вы должны явно добавить его в репозиторий. Такой осознанный файл позволяет разделить рабочие файлы от важных файлов.

Команда `git add <файл>` используется для добавления файла <файл> в репозиторий:

```
$ git add index.html
```

Примечание. Если в вашем каталоге есть несколько файлов, не нужно добавлять все их вручную, пусть за вас это сделает git. Чтобы добавить в репозиторий все файлы в каталоге и во всех подкаталогах, используйте команду git add . (одна точка в Unix означает текущий каталог).

После добавления файла Git знает, что файл index.html принадлежит репозиторию. Но Git просто подготовил файл, это еще не все. В Git разделены операции добавления и фиксации. Это сделано для лучшей производительности – вы только представьте, сколько времени займет обновление репозитория при каждом добавлении или удалении файла. Вместо этого Git предлагает раздельные операции, что особенно удобно в пакетном режиме.

Команда *git status* покажет промежуточное состояние файла index.html:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: index.html
```

Данная команда сообщает, что при следующей фиксации в репозиторий будет добавлен новый файл index.html.

Помимо актуальных изменений в каталоге и его файлах при каждой фиксации Git записывает различные метаданные, включая сообщение журнала и автора изменения. Полная команда фиксации выглядит так:

```
$ git commit -m "Начальное содержимое public_html"
--author="Jon Loeliger <jdl@example.com>"
Created initial commit 9da581d: Initial contents of public_
html
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 index.html
```

Вы можете предоставить сообщение журнала из командной строки, но обычно удобнее создать его с помощью интерактивного текстового редак-

тора. Чтобы Git открывал ваш любимый текстовый редактор при фиксации изменений, установите переменную окружения GIT_EDITOR:

```
# В tcsh
$ setenv GIT_EDITOR emacs
# В bash
$ export GIT_EDITOR=vim
```

После фиксации добавления нового файла в репозиторий, команда *git status* покажет, что нет ничего, что требовало ли фиксации:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Git также сообщает вам, что ваш рабочий каталог чист. А это означает, что в вашем рабочем каталоге нет неизвестных или измененных файлов, которые отличаются от тех, которые находятся в репозитории.

НЕЯСНЫЕ СООБЩЕНИЯ ОБ ОШИБКАХ

Git пытается определить автора каждого изменения. Если вы не сконфигурировали свое имя и e-mail так, чтобы они были доступны в Git, это может стать причиной некоторых предупреждений. Но не нужно паниковать, когда вы видите примерно такие сообщения:

```
You don't exist. Go away!
Your parents must have hated you!
Your sysadmin must hate you!
```

Они просто означают, что Git не может определить ваше реальное имя по какой-то причине. Проблема может быть исправлена путем установки вашего имени и e-mail, что будет показано в разделе "Настройка автора фиксации".

НАСТРОЙКА АВТОРА ФИКСАЦИИ

Перед фиксацией вам нужно установить некоторые параметры конфигурации и переменные окружения. Как минимум, Git должен знать ваше имя и ваш e-mail. Вы можете указывать эти данные при каждой команде фиксации (*git commit*), как было показано ранее, но это очень неудобно. Вместо этого лучше всего сохранить данную информацию в файле конфигурации, используя команду *git config*:

```
$ git config user.name "Jon Loeliger"
$ git config user.email "jdl@example.com"
```

Вы также можете сообщить Git свое имя и свой e-mail, установив переменные окружения GIT_AUTHOR_NAME и GIT_AUTHOR_EMAIL. Если эти переменные окружения установлены, они перезапишут все параметры, установленные в файле конфигурации.

ВНОСИМ ДРУГУЮ ФИКСАЦИЮ

Чтобы продемонстрировать еще несколько функций Git давайте внесем некоторые изменения и создадим сложную историю изменений внутри репозитория.

Откройте index.html, конвертируйте его в HTML, сохраните изменения и выполните фиксацию изменений:

```
$ cd ~/public_html

# редактируем index.html

$ cat index.html
<html>
<body>
Мой сайт жив!
</body>
</html>

$ git commit index.html
```

Когда откроется редактор, введите описание фиксации, например, "Файл конвертирован в HTML". После чего в репозитории появится две версии файла index.html.

ПРОСМОТР ВАШИХ ФИКСАЦИЙ

Как только в вашем репозитории появилось несколько фиксаций, вы можете исследовать их различными способами. Некоторые команды Git показывают последовательность отдельных фиксаций, другие – сводку об отдельной фиксации, а остальные показывают полную информацию о любой фиксации в каталоге.

Команда *git log* показывает историю отдельных фиксаций в репозитории:

```
$ git log
commit ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
Author: Jon Loeliger <jdl@example.com>
Date: Wed Apr 2 16:47:42 2008 -0500

Файл конвертирован в HTML

commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date: Thu Mar 13 22:38:13 2008 -0500

Начальное содержимое public_html
```

Записи выводятся в порядке от самой последней до самой старой. Каждая запись показывает имя и e-mail автора фиксации, дату фиксации и сообщение, добавленное в журнал при фиксации.

Для получения более подробной информации о конкретной фиксации, используйте команду *git show*:

```
$ git show 9da581d910c9c4ac93557ca4859e767f5caf5169
commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date: Thu Mar 13 22:38:13 2008 -0500
Initial contents of public_html
```

```
diff --git a/index.html b/index.html
new file mode 100644
index 0000000..34217e9
--- /dev/null
+++ b/index.html
@@ -0,0 +1 @@
+Мой сайт жив!
```

Если вы запустите *git show* без указания идентификатора фиксации, вы получите информацию о последней фиксации.

Другая команда, *show-branch*, выводит краткую сводку для текущей ветки разработки:

```
$ git show-branch --more=10
[master] Файл конвертирован в HTML
[master^] Начальное содержимое public_html
```

Параметр --more=10 указывает, что нужно вывести не более 10 версий, но в нашем случае пока существуют только две версии. Имя *master* – это имя по умолчанию для ветки.

ПРОСМОТР РАЗНИЦЫ МЕЖДУ ФИКСАЦИЯМИ

Чтобы увидеть разницу между двумя версиями index.html, вам понадобятся идентификаторы этих двух фиксаций, которые нужно передать команде *git diff*:

```
$ git diff 9da581d910c9c4ac93557ca4859e767f5caf5169 \
ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
diff --git a/index.html b/index.html
index 34217e9..8638631 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,5 @@
+<html>
+<body>
Мой сайт жив!
+</body>
```

```
+</html>
```

Этот вывод должен выглядеть знакомым. Он напоминает вывод команды *diff*. Первая версия, 9da581d910c9c4ac93557ca4859e767f5caf5169, является более ранней версией содержимого, а версия с именем ec232cddfb94e0dfd5 b5855af8ded7f5eb5c90d6 является более новой. Таким образом, знак "плюс" (+) предшествует каждой строке нового содержимого.

Вы испугались, глядя на эти шестнадцатеричные идентификаторы? Не волнуйтесь, Git предлагает более лаконичные способы идентифицировать фиксации без необходимости указания таких сложных чисел.

УДАЛЕНИЕ И ПЕРЕИМЕНОВАНИЕ ФАЙЛОВ В ВАШЕМ РЕПОЗИТОРИИ

Удаление файла из репозитории аналогично добавлению файла, но вместо команды *git add* используется команда *git rm*. Представим, что в нашем каталоге веб-сайта есть файл poem.html, который больше не нужен.

```
$ cd ~/public_html
$ ls
index.html poem.html

$ git rm poem.html
rm 'poem.html'

$ git commit -m "Удаляем поэму"
Created commit 364a708: Удаляем поэму
0 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 poem.html
```

Как и добавление, удаление состоит из двух этапов. Сначала *git rm* удаляет файл из репозитория, а затем *git commit* фиксирует изменения в репозитории. Опция -m позволяет указать описание фиксации, например, "Удаляем поэму". Вы можете опустить эту опцию, чтобы задать сообщение в вашем любимом текстовом редакторе.

Для переименования файла можно использовать комбинацию *git rm* и *git add* или же, что намного быстрее, использовать команду *git mv*:

```
$ mv foo.html bar.html
$ git rm foo.html
rm 'foo.html'
$ git add bar.html
```

В данном случае мы сначала переименовываем файл foo.html в bar.html и удаляем файл foo.html, как из репозитория, так и с файловой системы, после чего добавляем файл bar.html в репозиторий.

Эту же операцию можно выполнить с помощью одной команды:

```
$ git mv foo.html bar.html
```

После всего этого нам нужно фиксировать изменения:

```
$ git commit -m "Переименование foo в bar"
Created commit 8805821: Переименование foo в bar
1 files changed, 0 insertions(+), 0 deletions(-)
rename foo.html => bar.html (100%)
```

Git обрабатывает операции перемещения файла иначе, чем другие системы, используя механизм, основанный на базе подобия содержания между двумя версиями файлов.

КОПИРОВАНИЕ ВАШЕГО РЕПОЗИТОРИЯ

Ранее мы создали наш начальный репозиторий в каталоге ~/public_html. Теперь мы можем создать полную копию (или клон) этого репозитория командой *git clone*.

Сейчас мы создадим копию нашего репозитория в нашем домашнем каталоге, она будет называться my_website:

```
$ cd ~
$ git clone public_html my_website
```

Хотя эти два репозитория Git теперь содержат те же объекты, файлы и каталоги, есть некоторые тонкие различия. Исследовать эти различия можно командами:

```
$ ls -lsa public_html my_website
$ diff -r public_html my_website
```

На локальной файловой системе, подобно этой, использование *git clone* создаст копию репозитория подобно командам cp -a или rsync. Однако Git поддерживает богатый набор любых других источников, в том числе сетевые репозитории.

Как только вы клонируете репозиторий, вы сможете изменять, делать новые фиксации, исследовать журналы и историю и т.д. Это полностью новый репозиторий со своей полной историей.

ФАЙЛЫ КОНФИГУРАЦИИ

Конфигурационные файлы Git – это простые текстовые файлы в стиле ini-файлов. В них записываются различные установки, используемые многими Git-командами. Некоторые установки представляю собой сугубо личные предпочтения (например, color.pager), другие жизненно важны для правильного функционирования репозитория (core.repositoryformatversion), а остальные управляют поведением команд (например, gc.auto).

Подобно многим другим утилитам Git поддерживает иерархию конфигурационных файлов. В порядке уменьшения приоритета приведены его конфигурационные файлы:

- .git/config – специфичные для репозитория параметры конфигурации, которыми управляет опция --file. У этих настроек наивысший приоритет.
- ~/.gitconfig – специфичные для пользователя параметры конфигурации, которыми управляет опция --global.
- /etc/gitconfig – системные параметры конфигурации, изменить которые можно опцией --system, если у вас есть надлежащие права доступа Unix (нужно право на запись этого файла). Данные настройки обладают наименьшим приоритетом.

В зависимости от вашей инсталляции, системные настройки могут быть где-то еще, возможно, в /usr/local/etc/config, или могут полностью отсутствовать.

Например, для установки имени и e-mail пользователя, произведшего фиксацию, нужно указать значения параметров user.name и user.email в вашем файле$HOME/.gitconfig. Для этого используется команда git config --global:

```
$ git config --global user.name "Jon Loeliger"
$ git config --global user.email "jdl@example.com"
```

Можно установить специфичные для репозитория имя пользователя и адрес электронной почты, которые переопределят глобальные установки, для этого просто опустите флаг --global:

```
$ git config user.name "Jon Loeliger"
$ git config user.email "jdl@special-project.example.org"
```

Используйте git config -l для вывода всех переменных, найденных во всем наборе файлов конфигурации:

```
# Создаем пустой репозиторий
$ mkdir /tmp/new
$ cd /tmp/new
$ git init

# Устанавливаем некоторые переменные
$ git config --global user.name "Jon Loeliger"
$ git config --global user.email "jdl@example.com"
$ git config user.email "jdl@special-project.example.org"

$ git config -l
user.name=Jon Loeliger
user.email=jdl@example.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
user.email=jdl@special-project.example.org
```

Поскольку файлы конфигурации – это обычные текстовые файлы, вы можете просмотреть их содержимое командой *cat* и отредактировать в любимом текстовом редакторе:

```
# Посмотрим на параметры репозитария
$ cat .git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[user]
email = jdl@special-project.example.org
```

Или, если вы используете ОС от Microsoft, у вас будут некоторые изменения. Скорее всего, ваш конфигурационный файл будет выглядеть примерно так:

```
[core]
repositoryformatversion = 0
filemode = true
bare = true
logallrefupdates = true
symlinks = false
ignorecase = true
hideDotFiles = dotGitOnly
```

Большинство из этих изменений – из-за разницы в характеристиках файловой системы. Отключены символические ссылки, включено игнорирование регистра символов, другие настройки для скрытых файлов.

Используйте опцию --unset для удаления настройки:

```
$ git config --unset --global user.email
```

Поведение команды *git config* изменено между версиями 1.6.2 и 1.6.3. Более ранние версии потребовали, чтобы после опции --unset следовала опция --global. В новых версиях допустим любой порядок.

Для одной и той же цели существуют несколько параметров конфигурации и переменных окружения. Например, редактор, который будет вызван при создании сообщения журнала фиксации, можно задать с помощью:

- Переменной окружения GIT_EDITOR
- Опции конфигурации core.editor
- Переменной окружения VISUAL
- Переменной окружения EDITC
- Команды vi

Есть несколько сотен параметров конфигурации. По мере чтения книги я буду обращать ваше внимание на самые важные из них. На странице руководства (man) по команде *git config* вы найдете более подробный (и все же не полный) список параметров конфигурации.

НАСТРОЙКА ПСЕВДОНИМОВ

Новичкам пригодится совет по настройке псевдонимов командной строки. Часто команды Git довольно сложны, поэтому для самых часто используемых команд вы можете создать псевдонимы:

```
$ git config --global alias.show-graph \
'log --graph --abbrev-commit --pretty=oneline'
```

В этом примере я создал псевдоним show-graph и сделал его доступным в любом созданном мной репозитории. Теперь, когда я использую команду *git show-graph* будет выполнена длинная команда *git log* со всеми заданными опциями.

21.5. Основные понятия Git

В предыдущем разделе было рассмотрено типичное применение Git. Скорее всего, после ее прочтения у вас появилось гораздо больше вопросов, чем было до того. Какие данные Git хранит для каждой фиксации? Каково

назначение каталога .git? Почему ID фиксации напоминает мусор? Нужно ли мне обращать внимание на него?

Если вы использовали другую СКВ, например, SVN или CVS, команды, представленные ранее, наверняка покажутся вам знакомыми. На самом деле, Git служит для той же цели и предоставляет все операции, которые вы ожидаете увидеть в современной СКВ. Однако, Git некоторые понятия Git отличаются, о чем мы и поговорим.

Сейчас мы выясним, чем отличается Git от других СКВ, исследуя ключевые компоненты его архитектуры и некоторые важные понятия.

21.5.1. РЕПОЗИТОРИИ

Репозиторий Git – просто база данных, содержащая всю информацию, необходимую для управления версиями и историей проекта. В Git, как и в большинстве других систем управления версиями, репозиторий хранит полную копию всего проекта на протяжении всей его жизни. Однако, в отличие от большинства других VCS, репозиторий Git не только предоставляет полную рабочую копию всех файлов в репозитории, но также и копию самого репозитория, с которым работает.

В каждом репозитории Git обслуживает ряд значений конфигурации. Вы уже видели некоторые из них (имя пользователя и его e-mail) ранее. В отличие от данных файла и других метаданных репозитория, параметры конфигурации не переходят из одного репозитория в другой при клонировании. Вместо этого Git исследует конфигурацию и информацию об установке на основе пользователя, сайта, репозитория.

В репозитории Git управляет двумя основными структурами данных – хранилищем объектов и индексом. Все эти данные репозитория хранятся в корне вашего рабочего каталога в скрытом подкаталоге с именем .git.

Хранилище объектов разработано для эффективного копирования при операции клонирования как часть механизма, полностью поддерживающего распределенный VCS. Индекс – это переходная информация, персональная для репозитория. Индекс может быть создан или изменен по требованию в случае необходимости.

В следующих двух разделах подробно рассмотрены хранилище объектов и индекс.

21.5.2. ТИПЫ ОБЪЕКТОВ GIT

Сердце репозитория Git – это **хранилище объектов**. Оно содержит ваши исходные файлы и все сообщения журналов, информацию об авторе, даты и другую информацию, необходимые для сборки любой версии или ветки проекта.

В хранилище объектов Git помещает объекты четырех типов: *блобы* (от англ. *BLOB, Binary Large Object*), *деревья*, *фиксации* и *теги*.

БЛОБЫ

Каждая версия файла представлена как BLOB. **BLOB** – это аббревиатура для *"Binary Large Object"*, то есть большой двоичный объект. Этот термин часто используется в информатике для обозначения некоторой переменной или файла, которая (который) может содержать любые данные и внутренняя структура которой (которого) игнорируется программой. Блоб содержит данные файлы, но не содержит какие-либо метаданные о файле, даже его имя.

ДЕРЕВЬЯ

Дерево представляет один уровень информации каталога. Оно записывает идентификаторы блобов, имена путей и немного метаданных для всех файлов в каталоге. Оно также может рекурсивно ссылаться на другие поддеревья. Деревья используются для построения полной иерархии файлов и подкаталогов.

21.5.3. ФИКСАЦИИ

Объекты фиксации хранят метаданные для каждого изменения, произошедшего в репозитории, включая имя автора, дату фиксации и сообщение

журнала. Каждая фиксация указывает на объект дерева, который захватывает в одном полном снимке состояние репозитория на момент осуществления фиксации. У начальной (корневой) фиксации нет родителя. У большинства обычных фиксаций есть одна родительская фиксация, хотя фиксация может ссылаться на несколько родительских фиксаций.

ТЕГИ

Объект тега назначает человекочитаемое имя определенному объекту, обычно фиксации. Хотя 9da581d910c9c4ac93557ca4859e767f5caf5169 позволяет точно идентифицировать фиксацию, для человека более удобным идентификатором является что-то вроде Ver-1.0-Alpha.

Со временем вся информация в хранилище объектов изменяется и растет, отслеживая и моделируя ваши правки проекта, добавления и удаления файлов. Чтобы эффективно использовать дисковое пространство и пропускную способность, Git сжимает объекты в pack-файлы, которые также помещаются в хранилище объектов.

21.5.4. ИНДЕКС

Индекс – временный и динамический двоичный файл, который описывает структуру каталогов всего репозитория. В частности, индекс получает версию полной структуры проекта в некоторый момент времени. Состояние проекта может быть представлено фиксацией и деревом.

Одна из ключевых отличительных функций Git – то, что он позволяет вам изменить содержание индекса четко определенными действиями. Индекс позволяет разделение между шагами поэтапной разработки и поддержкой тех изменений.

Вот как это работает. Как разработчик, вы выполняете команды Git, чтобы подготовить изменения в индексе. Изменения обычно включают добавление, удаление или редактирование файлов. Индекс записывает эти изменения и хранит их, пока вы не зафиксируете их. Вы можете также удалить или заменить изменения в индексе. Таким образом, индекс – это как мост между двумя сложными состояниями репозитория.

21.5.5. АССОЦИАТИВНЫЕ ИМЕНА

Хранилище объектов Git организовано и реализовано как ассоциативная система хранения. В частности, каждому объекту в хранилище присваивается уникальное имя, применяя алгоритм SHA1 к содержанию объекта,

что приводит к значению хэш-функции SHA1. Поскольку значение хэш-функции вычисляется на основании содержания объекта, полагают, что оно уникально для определенного содержания. Следовательно, у каждого объекта будет уникальное имя в базе объектов. Крошечное изменение в файле приведет к изменению хэша SHA1, в итоге новая версия файла будет индексироваться отдельно.

Значения SHA1 – 160-разрядные значения, которые обычно представляются как 40-разрядное шестнадцатеричное число, такое как 9da581d910c9c4ac93557ca4859e767f5caf5169. Иногда во время отображения значения SHA1 сокращаются до меньшего уникального префикса. Пользователи Git называют эти числа SHA1, хэш-код и идентификатор объекта – все эти понятия взаимозаменяемы.

21.5.6. ГЛОБАЛЬНО УНИКАЛЬНЫЕ ИДЕНТИФИКАТОРЫ

Важная характеристика вычисления хеша SHA1 – то, что он всегда вычисляет тот же ID для идентичного содержания, независимо от того, где это содержание находится. Другими словами, одно и то же содержание файла в разных каталогах и даже на разных машинах даст одно и то же значение хэша SHA1. Таким образом, идентификатор SHA1 – это глобально уникальный идентификатор.

Благодаря глобально уникальным идентификаторам мы можем через Интернет сравнивать блобы или файлы произвольного размера путем простого сравнения их идентификаторов. Нам не нужно пересылать файлы по сети, чтобы определить, идентичны ли они.

21.5.7. GIT ОТСЛЕЖИВАЕТ КОНТЕНТ

Git – это нечто больше, чем просто СКВ. Git – это *система отслеживания контента*. Отслеживания контента в Git реализовано двумя способами, которые существенно отличаются от почти всех других систем управления версиями.

Во-первых, хранилище объектов Git основано на вычислении хэша содержимого объекта, а не на имени файла или каталога. Таким образом, когда Git помещает файл в хранилище, он это делает на основании данных хэша, а не на основании имени файла. Фактически, Git вообще не отслеживает име-

на файлов и каталогов, которые связаны с файлами вторичными способами. Git вместо имен файлов отслеживает их содержимое.

Если у двух отдельных файлов есть одинаковое содержание, независимо от того, хранятся ли они в одном каталоге или разным, Git хранит только одну копию этого содержания в виде блоба в хранилище объекта. Git вычисляет хэш-код каждого файла исключительно по его содержанию. Оба файла в проекте с одинаковым содержанием, независимо от того, где они расположены, используют один и тот же объект содержания (блоб).

Если один из этих двух файлов будет изменен, Git вычислит новый хэш SHA1 для него и добавит новый блоб в хранилище объектов. Исходный блоб останется в хранилище и будет использован для файла, который остался без изменений.

Во-вторых, внутренняя база данных Git хранит каждую версию каждого файла, а не разницу между ними. Поскольку Git использует хеш полного содержания файла как имя для этого файла, он должен оперировать с полной копией файла. Он не может основывать свою работу на части содержания файла или на разнице между версиями файла.

Типичное пользовательское представление файла в виде версий и изменений между версиями является обычным артефактом. Git вычисляет эту историю как ряд изменений между различными блобами вместо того, чтобы хранить имя файла и набор различий между версиями.

21.5.8. ПУТЬ ПО СРАВНЕНИЮ С СОДЕРЖАНИЕМ

Как и в случае с другими СКВ, Git должен вести явный список файлов, которые формируют содержание репозитория. Однако это не требует, чтобы внутренне Git основывался на именах файлов. Действительно, Git обрабатывает имя файла только как часть данные, которые отличны от содержания этого файла. В результате индекс отделяется от данных в традиционном смысле базы данных. Посмотрите на таблицу 21.1, которая сравнивает Git с другими известными системами.

Таблица 21.1. Сравнение баз данных

Система	Механизм индекса	Хранилище данных
Традиционная база данных	Индексно-Последовательный Метод Доступа (ISAM)	Записи данных
Файловая система UNIX	Каталоги (/путь/к/файлу)	Блоки данных
Git	.git/objects/hash, содержимое объекта дерева	Объекты блоб, объекты дерева

Названия файлов и каталогов происходят от базовой файловой системы, но Git действительно не заботится об именах. Он просто записывает каждый путь и удостоверяется, что он может точно воспроизвести файлы и каталоги из содержимого, которое индексировано значением хэша.

Физический формат данных не моделируется после пользовательской структуры каталога. Вместо этого есть абсолютно другая структура, которая может, тем не менее, воспроизвести оригинальную разметку пользователя. Внутренняя структура Git – более эффективная структура данных для своих собственных внутренних операций и средств хранения.

Когда Git нужно создать рабочий каталог, он говорит файловой системе: "Привет! У меня есть большой блоб данных, которые я хочу поместить в файл с именем /каталог/файл. Сделай это как посчитаешь нужным". На что файловая система отвечает: "Я распознала строку, являющуюся набором имен подкаталогов, и я знаю, куда поместить твои блоб- данные! Спасибо!".

21.5.9. PACK-ФАЙЛЫ

Проницательный читатель может заметить: "Очень неэффективно хранить полное содержимое каждой версии каждого файла. Даже если содержимое сжато, все равно неэффективно хранить содержимое разных версий каждого файла. Почему, если добавляется всего одна строка в файл, Git сохраняет полное содержимое файла?".

Давайте попробуем разобраться. Git использует очень эффективный механизм хранения, называемый pack-файлом. Для создания упакованного файла Git сначала определяет местоположение файлов, содержание которых очень подобно и хранит полное содержание для одного из них. Затем он вычисляет различия или дельты между подобными файлы и хранит просто различия. Например, если вам нужно просто изменить или добавить одну строку в файл, Git может сохранить полную, более новую версия файла и затем записать изменение одной строки как дельта и тоже сохранить ее в пакете.

Хранение полной версии файла и дельт, необходимых для создания других версий подобных файлов – далеко не новый прием. Это, по существу, тот же механизм, которые другие VCS, такие как RCS, использовали на протяжении многих десятилетий.

Git очень умно упаковывает файл. Поскольку Git основывается на контенте файла, ему действительно все равно, принадлежат ли дельты, которые он вычисляет между двумя файлами двум версиям одного и того же файла или нет. То есть Git может взять два любых файла и вычислить дельты между ними, если он считает, что они могли бы быть достаточно подобными, чтобы быть хорошо сжатыми. Таким образом, Git обладает тщательно продуманным алгоритмом, чтобы найти потенциальных кандидатов дельты по всему репозиторию.

Упакованные файлы хранятся в хранилище объектов вместе с другими объектами. Они также используются для передачи данных репозитория по сети.

21.5.10. ПОНЯТИЯ GIT В ДЕЙСТВИИ

Теперь, когда вы знакомы с некоторыми понятиями, давайте посмотрим,

как они работают в самом репозитории. Давайте создадим новый репозиторий и посмотрим на него внутренние файлы.

ВНУТРИ КАТАЛОГА .GIT

Для начала инициализируем пустой репозиторий, используя *git init*, а затем запустим команду *find*, чтобы посмотреть, что же было создано.

```
$ mkdir /tmp/hello
$ cd /tmp/hello
$ git init
Initialized empty Git repository in /tmp/hello/.git/

# Выводим все файлы в текущем каталоге
$ find .
.
./.git
./.git/hooks
./.git/hooks/commit-msg.sample
./.git/hooks/applypatch-msg.sample
./.git/hooks/pre-applypatch.sample
./.git/hooks/post-commit.sample
./.git/hooks/pre-rebase.sample
./.git/hooks/post-receive.sample
./.git/hooks/prepare-commit-msg.sample
./.git/hooks/post-update.sample
./.git/hooks/pre-commit.sample
./.git/hooks/update.sample
./.git/refs
./.git/refs/heads
./.git/refs/tags
./.git/config
./.git/objects
./.git/objects/pack
./.git/objects/info
./.git/description
./.git/HEAD
./.git/branches
./.git/info
./.git/info/exclude
```

Как видите, в каталоге .git много чего есть. Файлы, выведенные на экран,

основаны на каталоге шаблона, который вы можете при желании изменить. В зависимости от используемой версии Git содержимое этого каталога может немного изменяться. Например, более старые версии Git добавляют "расширение" .sample к файлам в каталоге .git/hooks.

В целом, вам не нужно ни просматривать, ни управлять файлами из каталога .git. Эти "скрытые" файлы считаются частью инфраструктуры или конфигурации Git. Конечно, в Git есть набор команд по управлению этими скрытыми файлами, но вы будете редко их использовать.

Изначально каталог .git/objects (каталог для всех объектов Git) пуст, за исключением нескольких заполнителей:

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/info
```

Теперь давайте осторожно создадим простой объект:

```
$ echo "hello world" > hello.txt
$ git add hello.txt
```

Если вы введете "hello world" точно так же (без изменений в регистре символов или интервале между ними), ваш каталог объектов будет выглядеть примерно так:

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Все это выглядит довольно таинственным. Но на самом деле все просто и в следующем разделе вы поймете почему.

ОБЪЕКТЫ, ХЭШИ, БЛОБЫ

Когда Git создает объект для hello.txt, он не беспокоится об имени файла hello.txt. Его интересует только то, что в файле: последовательность из 12 байтов – строка "hello world" и символ новой строки. Git выполняет несколько операций на этом блобе, вычисляет его хеш SHA1 и помещает в хранилище объектов как файл, в качестве имени файла используется вычисленное значение хэша.

Хэш в этом случае – 3b18e512dba79e4c8300dd08aeb37f8e728b8dad. 160 битов хэша SHA1 соответствуют 20 байтам, а при отображении на экране в шестнадцатеричном виде – 40 байтов. Итак, наш контент сохранен как .git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad.

Git вставляет / после первых двух разрядов, чтобы повысить эффективность файловой системы. Производительность некоторых файловых си-

стем заметно падает, если поместить в один каталог слишком много файлов. А если превратит первый SHA1 байт в каталог, то это самый простой способ создать фиксированное разделение пространства имен для всех возможных объектов с равным распределением.

Чтобы показать, что Git действительно ничего не сделал с содержанием файла (это все еще строка "hello world"), вы можете использовать хэш файла, чтобы получить доступ к содержимому в любой момент:

```
$ git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
hello world
```

> ***Примечание.*** Git также знает, что 40 символов немного рискованно, чтобы их вводить вручную, поэтому он предоставил команду для поиска объектов по префиксу хэша:
>
> ```
> $ git rev-parse 3b18e512d
> 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
> ```

ФАЙЛЫ И ДЕРЕВЬЯ

Теперь, когда блоб "hello world" безопасно помещен в хранилище объектов, что произошло с его именем файла? Git был бы не очень полезен, если он не мог бы найти файлы по имени.

Как было упомянуто выше, Git отслеживает пути файлов через другой вид объектов – деревья. Когда вы используете *git add*, Git создает объект для содержания каждого добавленного вами файла, но он не создает объект для вашего дерева сразу же. Вместо этого он обновляет индекс. Индекс хранится в .git/index и используется для отслеживания пути файла и соответствующих блобов. Каждый раз, когда вы выполняете команды вроде *git add*, *git rm* или *git mv*, Git обновляет индекс, устанавливая новую информацию блоба и пути файла.

Создать объект дерева из вашего текущего индекса можно с помощью низкоуровневой команды *git write-tree*.

Для просмотра содержимого индекса введите следующую команду (на момент ее ввода индекс содержал только один файл – hello.txt):

```
$ git ls-files -s
100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0 hello.txt
```

Здесь мы видим, что файл hello.txt соответствует блобу 3b18e5.... Далее, давайте получим состояние индекса и сохраним его как объект дерева:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ find .git/objects
.git/objects
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Теперь у нас есть два объекта: объект "hello world" с ID 3b18e5 и новый объект, объект дерева, с ID 68aba6. Как видите, имя объекта соответствует каталогу и файлу в каталоге .git/objects.

Но как выглядит само дерево? Поскольку дерево – это тоже объект, подобно блобу, вы можете использовать ту же команду для его просмотра:

```
$ git cat-file -p 68aba6
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad hello.txt
```

Содержимое объекта просто интерпретировать. Первое число – 100644 – представляет атрибуты файла в восьмеричной системе, если вы работали с Unix, то вы с ними знакомы. Далее идет имя (3b18e5) объекта, а hello.txt – имя файла, связанное с блобом.

ПРИМЕЧАНИЕ ОТНОСИТЕЛЬНО ИСПОЛЬЗОВАНИЯ SHA1

Перед более подробным рассмотрением содержимого объекта дерева, давайте посмотрим на очень важную функцию SHA1-хэшей:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

Каждый раз, когда Вы вычисляете другой объект дерева для того же индекса, хэш SHA1 остается неизменным. Git не должен воссоздать новый объект дерева. Если вы вводите эти команды на своем компьютере, то вы должны увидеть те же хэши, что и приведенные в этой книге.

Хэш-функция – истинная функция в математическом смысле: для заданного ввода она всегда производит один и тот же вывод.

Это чрезвычайно важно. Например, если вы создаете то же самое содержание как другой разработчик, независимо от того, где или когда или как вы оба работаете, идентичный хэш – доказательство полной идентичности содержимого.

Но задержитесь на секунду. Разве SHA1 хэши не являются уникальными? Что произошло с триллионами людей с триллионами блобов в секунду, которые никогда не произведут одну единственную коллизию? Это частый источник недоразумений среди новых пользователей Git. Внимательно продолжайте читать далее, поскольку если вы сможете понять эту разницу, тогда все остальное в этой главе – просто.

Идентичные хэши SHA1 в этом случае *не считаются коллизией*. **Коллизия** – это если два разных объекта производят один и тот же хэш. Здесь же мы создали два отдельных экземпляра одного и того же содержимого, поэтому хэш одинаковый (у одного и того же контента всегда будет одинаковый хэш).

Git зависит от другого последствия хеш-функции SHA1: не имеет значения, *как* вы получили дерево, названное 68aba62e560c0ebc3396e8ae9335232cd93 a3f60. Если оно есть у вас, вы можете быть полностью уверены, что это – тот же древовидный объект, который, скажем, есть у другого читателя этой книги. Боб, возможно, создал дерево, комбинируя ***фиксации A*** и ***B*** от Дженни и ***фиксации C*** от Сергея, тогда как вы получили ***фиксацию A*** от Сью и обновление от Лакшми, которое комбинирует ***фиксации B*** и *C*. Результат – тот же, и это существенно упрощает распределенную разработку.

Если вас попросили найти объект 68aba62e560c0ebc3396e8ae9335232cd93a 3f60 и вы можете найти именно этот объект, поскольку SHA1 – криптографический хэш и вы можете быть уверены, что нашли именно те данные, по которым был создан хэш.

Также истинно: если вы нашли объект с определенным хэшем в вашем хранилище объектов, вы можете быть уверены, что у вас нет копии этого объекта. Хэш таким образом является надежной меткой или именем для объекта.

Но Git также полагается на что-то более важное, чем просто заключение. Рассмотрим самую последнюю фиксацию (или связанный с ней объект дерева). Поскольку она содержит, как часть ее контента, хэш ее родительский фиксаций и ее дерево содержит хэш всех его поддеревьев и блобов. А это означает, что хэш исходной фиксации однозначно определяет состояние целой структуры данных, которая основана на той фиксации.

Наконец, импликации нашего требования в предыдущем абзаце приводят к мощному использованию хеш-функции: она предоставляет эффективный способ сравнения двух объектов, даже очень больших и сложных структур данных без передачи этих объектов полностью.

ИЕРАРХИЯ ДЕРЕВЬЕВ

Хорошо иметь информацию относительно одного файла, как было показано в предыдущем разделе, но проекты обычно более сложны и содержат глубоко вложенные каталоги, которые со временем перестраиваются и перемещаются. Давайте посмотрим, как Git обработает создание нового каталога, в который мы поместим полную копию файла hello.txt:

```
$ pwd
/tmp/hello
$ mkdir subdir
$ cp hello.txt subdir/
$ git add subdir/hello.txt
$ git write-tree
492413269336d21fac079d4a4672e55d5d2147ac
$ git cat-file -p 4924132693
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad hello.txt
040000 tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60 subdir
```

В новом корневом дереве теперь есть два элемента: исходный файл hello.txt и новый подкаталог **subdir**, который выводится как *tree*, а не как *blob*.

Что же тут необычного? Посмотрите на имя объекта **subdir**. Это наш старый друг – 68aba62e560c0ebc3396e8ae9335232cd93a3f60!

Новое дерево для **subdir** содержит только один файл, hello.txt и этот файл содержит старый контент – строку "hello world". Поэтому дерево **subdir** в глазах Git выглядит так же, как и корневой каталог.

Теперь давайте посмотрим на каталог .git/objects и посмотрим, на что повлияло это новое изменение:

```
$ find .git/objects
.git/objects
.git/objects/49
.git/objects/49/2413269336d21fac079d4a4672e55d5d2147ac
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

У нас все еще есть три *уникальных* объекта: блоб со строкой "hello world"; дерево, содержащее hello.txt, в котором есть текст "hello world" плюс новая строка; второе дерево, которое содержит *другую* ссылку на hello.txt в первом дереве.

ФИКСАЦИИ

Следующий объект для обсуждения – **фиксации**. Теперь, когда hello.txt был добавлен с помощью *git add* и был произведен объект дерева командой *git writer-tree*, вы можете создать объект фиксации, используя следующие команды:

```
$ echo -n "Commit a file that says hello\n" \
| git commit-tree 492413269336d21fac079d4a4672e55d5d2147ac
3ede4622cc241bcb09683af36360e7413b9ddf6c
```

Результат будет примерно таким:

```
$ git cat-file -p 3ede462
tree 492413269336d21fac079d4a4672e55d5d2147ac
author Jon Loeliger <jdl@example.com> 1220233277 -0500
committer Jon Loeliger <jdl@example.com> 1220233277 -0500
Commit a file that says hello
```

Если вы выполняете примеры из этой книге на своем компьютере, вы, вероятно, обнаружили, что у объекта фиксации, сгенерированного вами, будет другое имя, которое отличается от приведенного в книге. Если вы до сих пор понимали все написанное, причина должна быть очевидной: это не та фиксация. Ведь фиксация содержит ваше имя и время создания фиксации, а эти данные в вашем случае будут отличаться. С другой стороны, у вашей фиксации есть то же дерево, что и в примерах.

Это то, почему объекты фиксации отделяются от их объектов дерева: разные фиксации часто относятся к одному и тому же дереву. Когда это происходит, Git достаточно умен, чтобы передать только новый объект фиксации, который обычно крошечный, вместо копирования всего дерева и всех блоб-объектов, которые, наверняка, гораздо больше.

В реальной жизни вы можете (и должны) вызвать команды *git write-tree* и *git commit-tree*, и потом просто использовать команду *git commit*. Вам не нужно помнить все команды, чтобы быть счастливым пользователем Git.

Основной объект фиксации довольно прост и содержит следующее:

- Имя объекта дерева, который фактически идентифицирует связанные файлы.
- Имя человека, создавшего новую версию (*author*) и время создания этой версии.
- Имя человека, который поместил новую версию (*commiter*) в репозиторий и время фиксации.
- Описание версии (сообщение о фиксации).

По умолчанию *author* и *commiter* – это один и тот же человек. Но существуют ситуации, когда это разные люди.

> ***Примечание.*** Вы можете использовать команду git show --pretty=fuller для просмотра дополнительной информации о заданной фиксации.

Объекты фиксации также хранятся в виде структуры графа, хотя эта структура полностью отличается от структур, которые используются объектами дерева. Когда вы производите новую фиксацию, вы можете указать одну или больше родительских фиксаций.

ТЕГИ

Наконец, мы добрались до последнего объекта, которым управляет Git – тег. Хотя в Git реализован тег только одного вида, на самом деле существует два вида тегов – *легковесный* и *аннотированный*.

Легковесный тег – это просто ссылка на объект фиксации и обычно он частный для репозитория. Эти теги не создают постоянный объект в хранилище объектов. Аннотированный тег более существенный и создает объект, содержащий сообщение, предоставленное вами и может содержать цифровую подпись, созданную с помощью GPG-ключа согласно RFC4880.

Git обрабатывает, как легковесные, так и аннотированные теги, но по умолчанию большинство тегов Git работают только с аннотированными тегами, поскольку считают их "постоянными" объектами.

Создать аннотированный тег с сообщением можно командой *git tag*:

```
$ git tag -m "Tag version 1.0" V1.0 3ede462
```

Увидеть объект тега можно командой *git cat-file -p*, но как узнать хэш SHA1 объекта тега? Чтобы найти его, используйте совет из "Объекты, хэши и блобы":

```
$ git rev-parse V1.0
6b608c1093943939ae78348117dd18b1ba151c6a
$ git cat-file -p 6b608c
object 3ede4622cc241bcb09683af36360e7413b9ddf6c
type commit
tag V1.0
tagger Jon Loeliger <jdl@example.com> Sun Oct 26 17:07:15 2008
-0500
Tag version 1.0
```

В дополнение к сообщению журнала и информации об авторе тег ссылается на объект фиксации (3ede462).

Git обычно тегирует объект фиксации, указывающий на объект дерева, охватывающее общее состояние всей иерархии файлов и каталогов в вашем репозитории.

Поведение Git не похоже на другие СКВ, где тег применяется отдельно к каждому файлу, а затем на основании коллекции этих теггированных фай-

лов воссоздается целая теггированная версия. Другие СКВ позволяют вам перемещать теги отдельных файлов, а Git требует создания новой фиксации, которая охватывает изменение состояние файла, тег которого был перемещен.

21.6. Управление файлами

Когда ваш проект на попечении системы контроля версий, вы редактируете файлы проекта в своем рабочем каталоге, а потом передаете свои изменения в ваш репозиторий для сохранности. Git работает также, но в нем есть промежуточный уровень между рабочим каталогом и репозиторием – **индекс**. Индекс используется для подготовки, или организации, изменений. Когда вы управляете своим кодом с помощью Git, вы редактируете изменения в своем рабочем каталоге, накапливаете их в своем индексе, а затем фиксируете то, что накопилось в индексе как один набор изменений.

Вы можете думать об индексе, как о ряде намеченных или предполагаемых модификаций. Вы добавляете, удаляете, перемещаете или редактируете файлы до точки фиксации, которая реализовывает накопленные файлы в репозитории. Большая часть важной работы фактически предшествует шагу фиксации.

> ***Примечание***. Помните, что фиксация – двухступенчатый процесс. Сначала вы подготавливаете изменения, а потом вы фиксируете изменения. Изменение, найденное в рабочем каталоге, но не в индексе, не подготовлено и не может фиксироваться. Для удобства Git позволяет вам комбинировать эти два шага, когда вам нужно добавить или изменить файл:
>
> ```
> $ git commit index.html
> ```
>
> Но если вы переместили или удалили файл, у вас не будет такой роскоши. Вам нужно будет выполнить два действия:
>
> ```
> $ git rm index.html
> $ git commit
> ```

Сейчас мы поговорим, как управлять индексом. Будет показано, как добавлять и удалять файлы из вашего репозитория, как переименовать файл и т.д.

21.6.1. ВСЕ ОБ ИНДЕКСЕ

Линус Торвальдс утверждал в списке рассылки Git, что вы не можете осознать всю мощь Git без понимания назначения индекса.

Индекс Git не содержит названий файла, он просто отслеживает то, что вы хотите фиксировать. Когда вы выполняете фиксацию, Git проверяет индекс, а не ваш рабочий каталог.

Несмотря на то, что многие высокоуровневые команды Git разработаны, чтобы скрыть детали индекса, все еще важно помнить об индексе и его состоянии.

Вы можете запросить состояние индекса в любое время командой *git status*. Она явно отображает файлы, подготовленные для фиксации (находящиеся в индексе). Также вы можете узнать о внутреннем состоянии Git командами вроде *git ls-files*.

Вероятно, вам пригодится команда *git diff*. Эта команда может вывести на экран два различных набора изменений: *git diff* выводит изменения, которые есть в рабочем каталоге, но которых нет в индексе; git diff -cached выводит изменения, которые уже подготовлены (находятся в индексе) и будут помещены в репозиторий при следующей фиксации.

21.6.2. КЛАССИФИКАЦИЯ ФАЙЛОВ В GIT

Git классифицирует ваши файлы на три группы: *отслеживаемые*, *игнорируемые* и *неотслеживаемые*.

ОТСЛЕЖИВАЕМЫЕ

Отслеживаемым называется файл, уже находящийся в репозитории или в индексе. Чтобы добавить файл *somefile* в эту группу, выполните команду *git add somefile*.

ИГНОРИРУЕМЫЕ

Игнорируемый файл должен быть явно объявлен невидимым или игнорируемым в репозитории (даже при том, что он может присутствовать в вашем рабочем каталоге). В вашем проекте может быть много игнорируемых файлов: ваши личные заметки, сообщения, временные файлы, вывод компилятора и большинство файлов, сгенерированных автоматически во вре-

мя сборки проекта. Git по умолчанию ведет список игнорируемых файлов, но вы можете настроить свой репозиторий для распознания других игнорируемых файлов. Чуть позже будет показано, как это сделать.

НЕОТСЛЕЖИВАЕМЫЕ

Неотслеживаемым является любой файл, не относящийся к любой из предыдущих двух категорий. Git просматривает весь набор файлов в вашем рабочем каталоге и вычитает из него отслеживаемые и игнорируемые файлы, в результате получается список неотслеживаемых файлов.

Давайте исследуем разные категории файлов, создав совершенно новый рабочий каталог и репозиторий:

```
$ cd /tmp/my_stuff
$ git init

$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to
track)

$ echo "Новые данные" > data

$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# data
nothing added to commit but untracked files present (use "git
add" to track)
```

Изначально нет никаких отслеживаемых или игнорируемых файлов, поэтому набор неотслеживаемых файлов пуст. Как только вы создадите данные, *git status* покажет один неотслеживаемый файл.

Редакторы и окружения сборки часто оставляют временные файлы среди вашего исходного кода. Такие файлы обычно не должны отслеживаться, как файлы исходного кода. Чтобы Git игнорировал файлы в каталоге, просто добавьте имя этого файла в специальный файл .gitignore.

```
# Вручную создадим файл, который будет игнорироваться
$ touch main.o
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# data
# main.o

$ echo main.o > .gitignore

$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# .gitignore
# data
```

Как видите, файл main.o был проигнорирован, но *git status* показал новый неотслеживаемый файл .gitignore. Хотя этот файл имеет специальное назначение в Git, он обрабатывается как любой другой обычный файл в пределах вашего репозитория. Пока .gitignore не будет добавлен, Git будет рассматривать его как неотслеживаемый.

Следующие разделы демонстрируют несколько способов изменения статуса отслеживаемого файла, а именно добавление файла в индекс и удаление его из индекса.

21.6.3. ИСПОЛЬЗОВАНИЕ GIT ADD

Команда *git add* организует файл, после следующей фиксации (*git commit*) такой файл будет добавлен в репозиторий. В терминологии Git файл будет неотслеживаемым, пока он не будет добавлен командой *git add*, что изменит его статус на отслеживаемый. Если команде *git add* передать имя каталога, все файлы и подкаталоги этого каталога будут добавлены рекурсивно.

Давайте продолжить пример из предыдущего раздела:

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# .gitignore
# data
# Track both new files.

$ git add data .gitignore

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: .gitignore
# new file: data
#
```

Первая команда *git status* показывает, что у нас есть два неотслеживаемых файла и напоминает, что для того, чтобы сделать файл отслеживаемым,

нужно просто использовать команду *git add*. После выполнения команды *git add* оба файла (data и .ignore) будут отслеживаться и будут подготовлены для помещения в репозиторий при следующей фиксации.

В терминах объектной модели Git добавление файла командой *git add* означает копирование файла в хранилище объектов и его индексирование. Организацию файла (*git add*) так же называют "кэшированием файла" или "помещением файла в индекс".

Вы можете использовать команду *git ls-files* для определения хэш-кодов для организованных файлов:

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0 .gitignore
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0 data
```

Большинство ежедневных изменений в вашем репозитории, вероятно, будут простыми правками. После редактирования и перед фиксацией изменений нужно запустить *git add* для обновления индекса, чтобы занести в него последнюю и самую высокую версию вашего файла. Если вы это не сделаете, у вас будут две разные версии файла: одна из хранилища объектов, на которую ссылается индекс, и другая – в вашем рабочем каталоге.

Чтобы продолжить пример, давайте изменим данные файлы так, чтобы они отличались от тех, которые есть в индексе. После сего, командой *git hash-object <файл>* вычислим и отобразим хэш новой версии файла.

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0 .gitignore
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0 data
# отредактируйте файл "data"...
$ cat data
Новые данные
Еще немного данных
$ git hash-object data
e476983f39f6e4f453f0fe4a859410f63b58b500
After the file is amended, the previous version of the file in
the object store and index
has SHA1 534469f67ae5ce72a7a274faf30dee3c2ea1746d. However,
the updated version
of the file has SHA1 e476983f39f6e4f453f0fe4a859410f63b58b500.
Let's update the
index to contain the new version of the file:
```

```
$ git add data
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0 .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0 data
```

Теперь индекс содержит обновленную версию файла. Снова, "**данные** файла были подготовлены (для фиксации)" или, проще говоря, "**данные** файла были *помещены в индекс*". Последняя фраза не очень точная, но зато более понятная.

Опция --interactive для команд *git add* или *git commit* может быть полезна, чтобы узнать, какие файлы вы бы хотели подготовить для фиксации.

21.6.3. НЕКОТОРЫЕ ЗАМЕЧАНИЯ ОТНОСИТЕЛЬНО ИСПОЛЬЗОВАНИЕ GIT COMMIT

ИСПОЛЬЗОВАНИЕ GIT COMMIT -ALL

Опция -a или --all команды *git commit* заставляет Git автоматически подготавливать все неподготовленные, прослеженные изменения файла, в том числе удаление отслеживаемых файлов из рабочей копии, перед осуществлением фиксации.

Давайте посмотрим, как это работает, установив несколько файлов с разными характеристиками подготовки:

```
# Устанавливаем тестовый репозитарий
$ mkdir /tmp/commit-all-example
$ cd /tmp/commit-all-example
$ git init
Initialized empty Git repository in /tmp/commit-all-example/.
git/
$ echo something >> ready
$ echo somthing else >> notyet
$ git add ready notyet
$ git commit -m "Setup"
[master (root-commit) 71774a1] Setup
2 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 notyet
create mode 100644 ready
# Изменяем файл "ready" и добавляем командой "git add" его в
индекс
$ git add ready
```

```
# Изменяем файл "notyet"
# редактируем "notyet"
# Добавляем новый файл в подкаталог, но не добавляем его в
репозиторий
$ mkdir subdir
$ echo Nope >> subdir/new
```

Используем *git status*, чтобы просмотреть изменения, требующие фиксации:

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified: ready
#
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
# modified: notyet
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# subdir/
```

Здесь индекс подготовлен для фиксации только одного файла – с именем *ready*, поскольку только этот файл был подготовлен (добавлен).

Однако, если вы запустите команду *git commit -all*, Git рекурсивно обойдет весь репозиторий, организует все известные, измененные файлы. В этом случае, когда ваш редактор представит шаблон сообщения фиксации, он должен указать, что измененный и известный файл notyet тоже будет фиксироваться:

```
# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
```

```
# modified: notyet
# modified: ready
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# subdir/
```

Наконец, поскольку каталог с именем subdir/ новый и в нем не находится ни один из файлов, даже опция --all не заставит его фиксироваться:

```
Created commit db7de5f: Some --all thing.
2 files changed, 2 insertions(+), 0 deletions(-)
```

Git рекурсивно обойдет репозиторий в поисках измененных и файлов. Полностью новый каталог subdir/ и всего его файлы не станут частью фиксации.

НАПИСАНИЯ СООБЩЕНИЙ ЖУРНАЛА ФИКСАЦИИ

Если в командной строке вы явно не указываете сообщение журнала, Git запустит редактор и предложит вам написать его. Будет запущен установленный в вашей конфигурации редактор.

Если вы выйдете из редакторе без сохранения, Git будет использовать пустое сообщение журнала. Если вы уже сохранили сообщение, но еще не вышли из редактора, вы можете удалить сообщение и опять сохранить – тогда Git тоже сохранит пустое сообщение фиксации.

21.6.4. ИСПОЛЬЗОВАНИЕ GIT RM

Команда *git rm*, как и ожидается, обратна для *git add*. Она удаляет файл из репозитория и рабочего каталога. Однако удаление файла может быть более проблематичным (если что-то пойдет не так), чем добавление файла. Поэтому Git относится к удалению файла с большей осторожностью.

Git может удалить файл только из индекса или одновременно из индекса и рабочего каталога. Git не может удалить файл только из рабочего каталога, для этого используется команда операционной системы *rm*.

Удаление файла из каталога и из индекса не удаляет историю файла из репозитория. Любая версия файла до момента удаления хранится в хранилище объектов и не будет оттуда удалена.

Давайте представим, что у нас есть некоторый файл, который мы еще не добавили в репозиторий (не выполнили команду *git add*), и мы пытаемся удалить его командой *git rm*:

```
$ echo "Random stuff" > oops
# Не можем удалить файл, не добавленный в репозиторий
# Нужно использовать команду "rm oops"
$ git rm oops
fatal: pathspec 'oops' did not match any files
```

Теперь давайте добавим файл командой *git add*, а затем выполним команду *get status*:

```
# Добавляем файл
$ git add oops
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: .gitignore
# new file: data
# new file: oops
#
```

Чтобы конвертировать файл из подготовленного в неподготовленный, используйте команду *git rm --cached*:

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0 .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0 data
100644 fcd87b055f261557434fa9956e6ce29433a5cd1c 0 oops
$ git rm --cached oops
rm 'oops'
$ git ls-files --stage
```

```
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0 .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0 data
```

Обратите внимание: *git rm --cached* удаляет файл только из индекса, но оставляет его в рабочем каталоге, в то время как команда *git rm* удаляет файл, как из индекса, так и с рабочего каталога.

> ***Примечание***. Использование команды git rm --cached делает файл неотслеживаемым, в то время как его копия остается в рабочем каталоге. Это довольно опасно, так как вы можете забыть о нем, и файл больше никогда не будет отслеживаемым. Будьте осторожны!

Если вы хотите удалить файл, как только он был зафиксирован, просто отправьте запрос через команду git rm <файл>:

```
$ git commit -m "Add some files"
Created initial commit 5b22108: Add some files
2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 .gitignore
create mode 100644 data
$ git rm data
rm 'data'
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# deleted: data
#
```

Перед удалением файла Git проверяет, соответствует ли версия файла в рабочем каталоге последней версии в текущем ответвлении (в версии, которую команды Git называют HEAD). Эта проверка устраняет случайную потерю любых изменений, которые, возможно, были сделаны в файле.

> ***Примечание***. Используйте команду git rm -f для принудительного удаления вашего файла. В этом случае файл будет удален, даже если он был изменен с момента последней фиксации.

Если вам нужно сохранить файл, который вы случайно удалили, просто добавьте его снова:

```
$ git add data
fatal: pathspec 'data' did not match any files
```

Ошибочка вышла! Git ведь удалил и рабочую копию тоже! Но не волнуйтесь, VCS содержит отличный механизм восстановления старых версий файлов:

```
$ git checkout HEAD -- data
$ cat data
Новые данные
Еще немного новых данных
$ git status
# On branch master
nothing to commit (working directory clean)
```

21.6.5. ИСПОЛЬЗОВАНИЕ GIT MV

Предположим, что вам нужно удалить или переименовать файл. Вы можете использовать комбинацию команд *git rm* (для удаления старого файла) и *git add* (для добавления нового файла). Или же вы можете использовать непосредственно команду *git mv*. Представим, что в нашем репозитории есть файл с именем *stuff* и вы хотите переименовать его в *newstuff*. Следующие две последовательности действий являются эквивалентными:

```
$ mv stuff newstuff
$ git rm stuff
$ git add newstuff
```

или

```
$ git mv stuff newstuff
```

В обоих случаях Git удалит путь *stuff* из индекса, добавит новый путь *newstuff*, сохраняя оригинальное содержимое *stuff* в хранилище объектов и реассоциирует это содержимое с путем *newstuff*.

Файл *data* мы уже восстановили, теперь давайте его переименуем в *mydata*:

```
$ git mv data mydata
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# renamed: data -> mydata
#

$ git commit -m "Moved data to mydata"
Created commit ec7d888: Moved data to mydata
1 files changed, 0 insertions(+), 0 deletions(-)
rename data => mydata (100%)
```

Если вы проверите историю файла, вы можете быть немного удивлены, когда увидите, что Git, очевидно, специально потерял историю исходного файла и помнит, только, что данные были переименованы:

```
$ git log mydata
commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Jon Loeliger <jdl@example.com>
Date: Sun Nov 2 19:01:20 2008 -0600
Moved data to mydata
```

Git все еще помнит всю историю, но отображает только то, что касается определенного имени файла, указанного в команде. Опция --follow просит Git отследить журнал и найти всю историю, связанную с контентом:

```
$ git log --follow mydata
commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Jon Loeliger <jdl@example.com>
Date: Sun Nov 2 19:01:20 2008 -0600

Moved data to mydata

commit 5b22108820b6638a86bf57145a136f3a7ab71818
Author: Jon Loeliger <jdl@example.com>
Date: Sun Nov 2 18:38:28 2008 -0600
```

```
Add some files
```

Одна из классических проблем многих VCS заключается в том, что после переименования файла невозможно отследить его историю. В Git эта проблема решена.

21.6.7. ЗАМЕЧАНИЕ ОТНОСИТЕЛЬНО ОТСЛЕЖИВАНИЯ ПЕРЕИМЕНОВАНИЙ

Давайте немного подробнее рассмотрим отслеживание переименований файла.

SVN, как пример традиционного управления версиями, производит большую работу по отслеживанию во время переименования/перемещения файла, поскольку она отслеживает только разницу между файлами. Если вы перемещаете файл, то это, по сути, то же, что и удаление всех строк из старого файла и их добавление в новый файл. Но передавать все содержание файла каждый раз, когда вы делаете простое переименование, очень неэффективно. Подумайте о том, что будет, если нужно переименовать целый подкаталог с тысячами файлов.

Чтобы облегчить эту ситуацию, SVN явно отслеживает каждое переименование. Если вы хотите переименовать hello.txt в subdir/hello.txt, вы должны использовать команду *svn mv* вместо команды *svn rm* и *svd add*. Иначе SVN никак не поймет, что это переименование и ему придется пойти по неэффективному пути удаления/добавления, что и было описано выше.

Затем, учитывая эту исключительную функцию отслеживания переименования, SVN нуждается в специальном протоколе, чтобы сказать его клиентам: "переместите файл hello.txt в subdir/hello.txt". Кроме того, каждый клиент SVN должен убедиться, что выполнил эту работу правильно.

Git, с другой стороны, не отслеживает переименование. Вы можете переместить или скопировать hello.txt куда угодно, но это влияет только на объекты дерева. Помните, что объекты дерева хранят отношения между содержимым, тогда как само содержимое хранится в блобах. Посмотрев на разницу между двумя деревьями, становится очевидным, что блоб с именем 3b18e5 переместился в новое места.

В этой ситуации, как и во многих других, система хранения Git, основанная на хэше, упрощает много вещей по сравнению с другой RCS.

ПРОБЛЕМЫ С ОТСЛЕЖИВАНИЕМ ПЕРЕИМЕНОВАНИЯ

Отслеживание переименования файла порождает постоянные дебаты среди разработчиков VCS.

Простое переименование – объект разногласия. Аргументы становятся еще более весомыми, когда изменяется и имя, и содержимое файла. Тогда сценарий переговоров переходит от практического к философскому. Что это: переименование или новый файл (раз у него другое содержимое и другое имя)? Насколько новый файл подобен старому? Если вы применяете чей-то патч, который удаляет файл и воссоздает подобный в другом месте, как это обрабатывать? Что произойдет, если файл переименован двумя различными способами на двух разных ветках? Какая тактика менее подвержена ошибкам: используемая в Git или в SVN?

В реальной жизни, похоже, что система отслеживания переименований, используемая в Git, очень хороша, поскольку есть много способов переименовать файл и человек может просто забыть уведомить SVN об этом. Но помните, что нет идеальной системы для обработки переименований, к сожалению.

21.6.8. ФАЙЛ .GITINGNORE

Ранее в этой главе было показано, как использовать файл .gitignore для игнорирования файла main.o. Чтобы проигнорировать любой файл, просто добавьте его имя в файл .gitignore, который находится в этом же каталоге. Вы также можете игнорировать файлы где угодно, добавив его в файл .gitignore, который находится в корневом каталоге вашего репозитория.

Но Git предоставляет более богатый механизм. Файл .gitignore может содержать список шаблонов имен файлов, указывающий, какие файлы нужно игнорировать. Формат .gitignore следующий:

- Пустые строки игнорируются, как и строки, начинающиеся с решетки (#). Такие строки можно использовать для комментариев, однако символ # не представляет комментарий, если он не является первым в строке.
- Обычные имена файлов соответствуют файлу в любом каталоге с указанным именем.

- Имя катала отмечается с помощью слеша (/). Это правило соответствует любому каталогу или любому подкаталогу, но не соответствует файлу или символической ссылке.
- Шаблон может содержать маски оболочки, такие как звездочка (*). Звездочка может соответствовать единственному имени файла или каталога. Также звездочка может быть частью шаблона, включающего наклонные черты для обозначения имен каталогов, например, debug/32bit/*.o.
- Восклицательный знак (!) инвертирует смысл шаблона оставшейся части строки. Дополнительно, любой файл, исключенный предшествующим образцом, но соответствующий этому правилу инверсии, будет включен. У инверсии более высокий приоритет.

Кроме того, Git позволяет вам создавать файл .gitignore в любом каталоге вашего репозитория. Каждый такой файл влияет на свой каталог и все подкаталоги. Правила .gitignore каскадные: вы можете переопределить правила в каталоге более высокого уровня, включив инвертированный шаблон (с использованием ! в начале правила) в одном из подкаталогов.

Приоритет игнорирования следующий:

- Шаблоны, определенные в командной строке
- Шаблоны, прочитанные из файла .gitignore, находящего в том же каталоге
- Шаблоны в родительских каталогах. Шаблоны, находящиеся в текущем каталоге, будут перезаписывать шаблоны родительского каталога, следовательно, шаблоны более близкого родителя перезапишут шаблоны родителя более высокого уровня.
- Шаблоны из файла .git/info/exclude
- Шаблоны из файла, указанного переменной конфигурации core. excludefile.

Поскольку .gitignore обрабатывается как обычный файл в вашем репозитории, он копируется во время операций клонирования и применяется ко всем копиям вашего репозитория.

Если образец исключения, так или иначе определенный для одного вашего репозитория, не должен применяться к какому-нибудь клону этого репозитория, то шаблоны должны находиться в файле .git/info/exclude, поскольку

он не копируется во время операций клонирования. Формат его шаблонов полностью совпадает с форматом файла .gitignore.

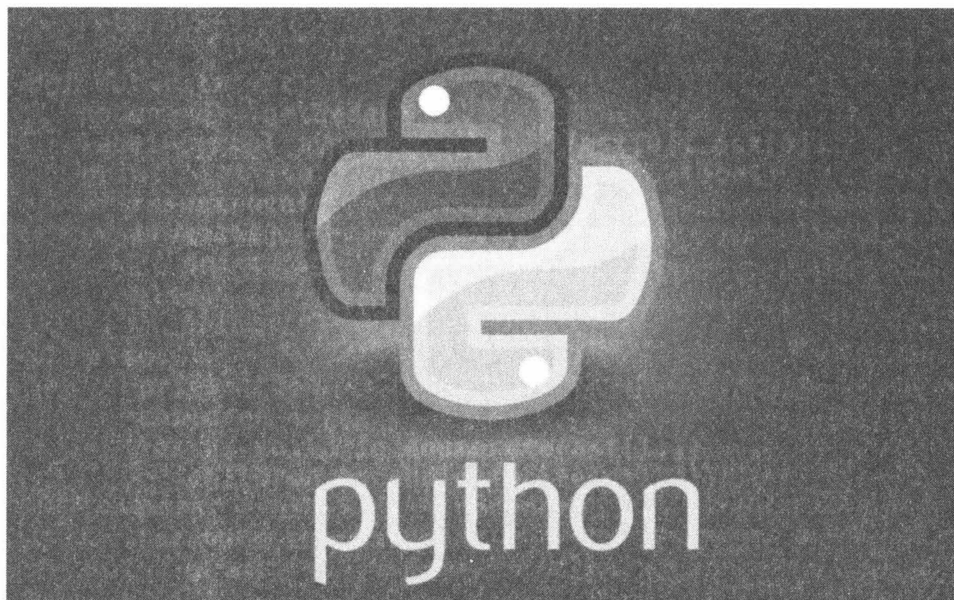
Рассмотрим другой сценарий. Нам нужно исключить файлы *.o, сгенерированные компилятором из исходного кода. Для игнорирования этих файлов поместите шаблон *.o в файл .gitignore самого верхнего уровня. Но что если в определенном каталоге у вас есть определенный .o файл, который нужно отследить? Тогда у вас может быть примерно эта конфигурация:

```
$ cd my_package
$ cat .gitignore
*.o
$ cd my_package/vendor_files
$ cat .gitignore
!driver.o
```

Данная конфигурация игнорирует все .o файлы в репозитории, но Git будет отслеживать одно исключение – файл driver.o в подкаталоге vendor_files.

ГЛАВА 22.

ОПТИМИЗАЦИЯ КОДА PYTHON



Профилирование используется для поиска узких мест в коде программы. Посредством профилирования разработчик или тестировщик может найти части кода, выполняющиеся дольше остальных. Далее, разработчик может оптимизировать эти части так, чтобы они выполнялись быстрее. В Python имеется три встроенных профайлера: *cProfile*, *profile* и *hotshot*. Последний использовать не рекомендуется – он устарел и уже не поддерживается. Модуль *profile* это в корне своем модуль Python, но добавляет много чего сверху в профилированные программы. Поэтому лучше использовать *cProfile*, который содержит интерфейс, имитирующий модуль *profile*.

22.1. Профилирование кода с помощью cProfile

Профилировать код с помощью *cProfile* достаточно просто. Вам нужно только импортировать модуль и вызвать команду *run*. Сразу рассмотрим простенький пример:

```
import hashlib
import cProfile
```

```
cProfile.run("hashlib.md5(b'hello').digest()")
```

Результат профилирования показан на рис. 22.1.

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import hashlib
>>> import cProfile
>>> cProfile.run("hashlib.md5(b'hello').digest()")
         5 function calls in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method _hashlib.openssl_
md5}
        1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'digest' of '_hashlib.HASH
' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof
iler' objects}


>>> |
Ln: 18 Col: 4
```

Рис. 22.1. Результат профилирования

Здесь мы импортировали модуль *hashlib* и использовали *cProfile* для профилирования того, что создал хеш MD5. Первая строка показывает, что в ней 5 вызовов функций. Следующая строка говорит нам, в каком порядке результаты выдачи. Здесь есть несколько столбцов.

- *ncalls* – это количество совершенных вызовов;
- *tottime* – все время, потраченное в данной функции;
- *percall* – ссылается на коэффициент tottime, деленный на ncalls;
- *cumtime* – совокупное время, потраченное как в данной функции, так и наследуемых функциях. Работает также и с рекурсивными функциями.

- Второй столбец *percall* – это коэффициент cumtime, деленный на примитивные вызовы;
- *filename:lineno(function)* предоставляет соответствующие данные о каждой функции.

Примитивный вызов – это вызов, который был совершен без рекурсии. Это очень интересный пример, так как здесь нет очевидных узких мест. Давайте создадим часть кода с узкими местами, и посмотрим, обнаружит ли их профайлер.

```
# -*- coding: utf-8 -*-
import time

def fast():
    print("Быстрая функция")


def slow():
    time.sleep(4)
    print("Медленная функция")


def medium():
    time.sleep(0.5)
    print("Средняя функция...")


def main():
    fast()
    slow()
    medium()

if __name__ == '__main__':
    main()
```

Сохраните программу как proftest.py. В этом примере нами создано четыре функции. Первые три работают с разными темпами. Быстрая функция запустится с нормальной скоростью, средняя функция потратит примерно полсекунды на запуск, медленная функция потратит примерно 5 секунд для запуска. Главная функция вызывает остальные три. Давайте запустим *cProfile* в этой простой программе:

```
import cProfile
import proftest

cProfile.run('proftest.main()')
```

Рис. 22.2. Результат профилирования

На этот раз мы видим, что у программы ушло 4.536 секунды на запуск. Если вы изучите результаты, то увидите, что *cProfile* выявил медленную функцию, которая тратит 4 секунды на запуск. Это и есть самая "слабая" часть основной функции. Обычно, когда вы обнаруживаете такие места, есть два варианта развития ситуации:

1. Вы приходите к заключению, что такая задержка приемлема.
2. Переписываете алгоритм (код), если это возможно.

В нашем простом примере, мы знаем, что лучший способ ускорить функцию, то убрать вызов *time.sleep*, или, по крайней мере, снизить продолжительность сна. На практике бывает сложно определить, как можно оптимизировать программу и часто оптимизация заключается в изменении алгоритма ее работы. При необходимости можно также вызвать *cProfile* в командной строке, вместо применения в интерпретаторе. Вот как это можно сделать:

```
python -m cProfile ptest.py
```

В этом случае *cProfile* будет запущен в вашем скрипте аналогично тому, как мы делали это ранее. Вам нужно сохранить выдачу профайлера? *cProfile* также позволяет это сделать. Все что вам нужно, это передать ему параметр –o, за которым следует название (или путь) файла. Пример:

```
python -m cProfile -o output.txt proftest.py
```

К сожалению, файл результата едва ли можно назвать читаемым. Для его чтения вам пригодится модуль Python pstats. Вы можете использовать *pstats* для форматирования выдачи разными способами. Разберем пример, демонстрирующий, как получить выдачу, по аналогии с тем, как мы делали это раньше:

```
import pstats
p = pstats.Stats("output.txt")
```

```
p.strip_dirs().sort_stats(-1).print_stats()
```

Вызов *strip_dirs* вырезает все пути к модулям из вывода, а вызов *sort_stats* делает сортировку, которая нужна нам для виденья картины. В документации по *cProfile* вы найдете множество интересных примеров, которые наглядно демонстрируют различные пути извлечения информации с использованием модуля *pstats*.

Конечно, можно использовать средства перенаправления ввода/вывода операционной системы:

```
python -m cProfile proftest.py > out.txt
```

В этом случае вы сразу получите читаемый текстовый файл.

22.2. Практический пример: вычисление скорости загрузки сайта

В данном примере мы пройдем небольшой марафон и узнаем, какой сайт быстрее всех откроется из нашей программы.

```
# -*- coding: utf-8 -*-
import requests
import cProfile

def facebook():
    requests.get('https://facebook.com')
def google():
    requests.get('https://google.com')
def twitter():
    requests.get('https://twitter.com')
def lenta():
    requests.get('https://lenta.ru')

def main():
    facebook()
    google()
    twitter()
```

```
    lenta()
cProfile.run('main()')
```

Далее мы сократили вывод профайлера, удалив все лишнее:

```
48556 function calls (48476 primitive calls) in 2.158 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
...
1     0.000     0.000     1.023     1.023 speedtest.py:10(google)
1     0.000     0.000     0.462     0.462 speedtest.py:12(twitter)
1     0.000     0.000     0.216     0.216 speedtest.py:14(lenta)
1     0.000     0.000     2.158     2.158 speedtest.py:18(main)
1     0.000     0.000     0.457     0.457 speedtest.py:6(facebook)
```

Самым медленным стал Google, затем – Twitter, чуть быстрее (хотя примерно такой же) – Facebook, а самый быстрый – Lenta.ru.

22.3. Событийные профайлеры

Событийные (*event-based*) профайлеры привязаны к событиям, как несложно догадаться из их названия. Статистические профайлеры срабатывают каждые N раз. А событийные профайлеры — только в определенных случаях, например, после вызова функции или завершения работы функции (смотря, к какому событию они привязаны).

Событийные профилировщики заметно точнее статистических, но точность здесь достигается в ущерб скорости работы. Из-за этого ними редко пользуются в продакшене. На Python мы можем написать свой профайлер – при желании:

```
import sys

def profiler(frame, event, args):
    print(frame.f_lineno, event, args)
```

```
sys.setprofile(profiler)

def main(name):
    print('Hello, %s!' % name)

if __name__ == '__main__':
    main('world')
```

Самое интересное здесь происходит после вызова функции sys.setprofile, которая говорит интерпретатору, что теперь у нас есть профайлер. После этого интерпретатор на каждое событие будет вызывать функцию *profiler*. В Python таких событий не так уж и много: *вызов функции* (*call*), *возврат из функции* (*return*) и *обработка исключения* (*exception*).

Функция-профайлер принимает три параметра:

- *frame* – представляет собой текущий стековый фрейм выполнения нашей программы (sys._current_frames);
- *event* – строка, имя события;
- *args* – специальные аргументы, которые отличаются в зависимости от типа события.

Классическим примером такого профайлера в Python служит *cProfile*, который является частью стандартной библиотеки Python и написан в виде C-расширения. В документации достаточно подробно написано о *cProfile*, поэтому не будем повторяться:

https://docs.python.org/3/library/profile.html

22.4. Ручное профилирование

Не стоит забывать также и о возможности ручного профилирования кода. Если мы уже нашли узкое место в коде, можно использовать так называемое ручное профилирование – способ профилирования, при котором мы руками вставляем код, измеряющий скорость работы. Все очень и очень просто – мы "засекаем" время до вызова функции, затем – после и таким нехитрым способом узнаем, сколько она выполнялась. Рассмотрим простой пример:

```
import time

t1 = time.time()
do_something()
t2 = time.time

print(t2-t1)
```

Одной из часто встречающихся реализаций такого профилировщика можно считать такой декоратор:

```
def profiler(func):
    def wrapper(*args, **kwargs):
        before = time.time()
        retval = func(*args, **kwargs)
        after = time.time()
        LOG.debug("Function '%s': %s", func.__name__, after-before)

    return wrapper

@profiler
def hello(name):
    print('Hello, %s' % name)
```

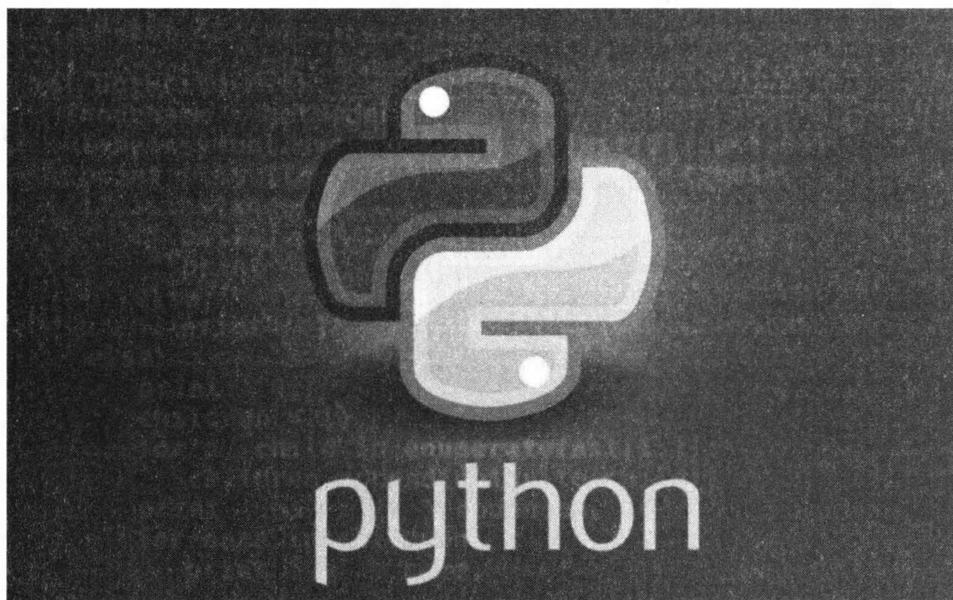
Решение настолько простое и эффективное, что в него даже нечего добавить. Тем не менее, следует помнить, что такой "профайлер" будет снижать скорость работы программы, и использовать его в продакшене крайне не рекомендуется.

Скорее всего, это самый простой способ измерить скорость выполнения какого-то фрагмента кода. Большой недостаток этого решения заключается в следующем: необходимо не просто модифицировать код — нужно писать его. А это означает, что с большой долей вероятности этот код будет нельзя переносить между проектами. Ведь такой код пишется не просто под нужды определенного проекта, а под нужды определенной функции или нескольких функций.

Зато есть и преимущество: можно измерять только то, что нам надо, и тогда, когда это необходимо. Например, включать профилирование только ночью, когда посетителей сайта мало и производительностью можно немного пожертвовать. Также можно включать такой "профайлер", если сработает какое-то условие ("пользователь нажал кнопку А"), и проверить скорость работы нужной функции.

ГЛАВА 23.

МНОГОЗАДАЧНОСТЬ В PYTHON



Многозадачность в Python наиболее часто проявляется в виде параллельной обработки и является одним из самых сложных вопросов в области программной инженерии. Данной теме можно посвятить целую книгу, да и не одну, поэтому в данной главе будут даны лишь поверхностные сведения, и, если вы заинтересуетесь, вы сможете продолжить изучение данной тематики в других источниках.

23.1. Есть ли необходимость в многозадачности?

Зачем нужна многозадачность и что это вообще такое? Начнем со второго вопроса, ответ на который может удивить тех разработчиков, которые считают, что многозадачность и параллельная обработка – это одно и то же.

Если попытаться вникнуть в теорию распределенных систем, то два события считаются параллельными, если ни одно из них не влияет на другое. Другими словами, мы можем сказать, что нечто выполняется одновременно, если его можно полностью или частично разложить на составные части, которые не зависят друг от друга. Такие части могут обрабатываться

независимо друг от друга и порядок их обработки не влияет на конечный результат.

Если мы обрабатываем информацию таким образом, то речь идет действительно о параллельной обработке. В качестве примера приведем программы по загрузке файлов, где файл разбивается на части и каждая часть загружается отдельно – в отдельном потоке, затем все эти части соединяются воедино в один файл. В этом случае действительно несколько потоков выполняются одновременно настолько, насколько это возможно (например, если в системе есть 1 процессор с одним ядром, то о каком параллельном выполнении может идти речь, если в определенный момент времени может выполняться инструкции только одного "потока"). Каждый из потоков загружает свою часть файла, а затем все это соединяется вместе, и нет разницы, в каком порядке были загружены эти части – главное, что на выходе вы получите целый и неповрежденный в результате такой загрузки файл.

По-настоящему выполнять задачи распределенным способом мы можем только при наличии многоядерных процессоров или вычислительных кластеров. В одно время (начало и первая половина 2000-х) кластеры были очень популярны для организации параллельных вычислений – тогда в кластер объединялось несколько машин, на которых выполнялось специальное ПО для параллельных вычислений. Сегодня, когда 8 ядерным (и более) процессором никого не удивишь, настоящую многозадачность можно реализовать не только на настольном компьютере, но и даже на смартфоне.

Теперь разберемся, зачем нам многозадачность. Естественно, для повышения эффективности работы приложения – обработка информации будет происходить быстрее. При разработке программ мы привыкли к последовательности выполнения шагов. Большинство компьютерных программ используют синхронные алгоритмы, позволяющие выполнять по одному действию за один раз. Такой способ обработки данных не очень подходит для масштабных задач или кейсов. К многозадачному выполнению приходят в следующих случаях:

1. Когда масштаб задачи настолько велик, что единственный способ решить ее за приемлемое время – это распределение ее выполнения на несколько блоков (потоков), которые могут обрабатывать задачу параллельно.
2. Когда есть необходимость принимать новые данные, даже если обработка старых еще не завершена.

Первая группа задач обычно решается с помощью многопоточных и многопроцессорных моделей. Вторая группа задач далеко не всегда требует

параллелизма и часто ее решение зависит от тонкостей самой задачи. Эта группа задач охватывает и кейс, когда приложение должно обслуживать несколько клиентов (например, есть веб-сервер, который обслуживает несколько разных пользователей) независимо друг от друга, не нуждаясь в ожидании успешного обслуживания других клиентов.

Обе эти группы не исключают друг друга. Часто нужно сохранить время отклика приложения и в то же время обрабатывать данные на одном процессоре. Поэтому, казалось бы, разные и конфликтующие подходы к параллельности можно использовать одновременно. Часто это сочетается в разработке веб-серверов, где нужно задействовать асинхронные циклы событий или потоки в сочетании с несколькими процессами, чтобы применять все доступные ресурсы и обеспечивать низкие задержки при высокой нагрузке.

23.2. Многопоточность

Реализация многопоточности – довольно сложная штука. Однако есть различные высокоуровневые классы и функции, облегчающие использование многопоточности в ваших приложений. Сразу нужно отметить, что реализация многопоточности в Python не такая удачная, как в C или Java. В этом разделе мы поговорим об ограничениях многопоточности в Python, а также рассмотрим общие задачи, где многопоточность жизненно необходима.

23.2.1. ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ

Первым делом нужно разобраться, что такое многопоточность. Разработчик приложения может разделить свою работу на потоки, работающие одновременно и использующие один и тот же контекст памяти. Если код не зависит от сторонних ресурсов, то многопоточность никак не позволит ускорить работу на одноядерном процессоре, а только лишь добавит лишние затраты ресурсов на управление потоками. Другими словами, для реализации многопоточности нужен как минимум двухядерный процессор, в противном случае многопоточность навредит производительности.

На многоядерных или многопроцессорных машинах, где каждый поток может выполняться на отдельном ядре процессора, многопоточность позволя-

ет повысить производительность программы. В принципе, данная аксиома характерна для всех языков программирования, а не только Python.

> ***Примечание.*** Раньше, скажем лет 20-25 назад, все было предельно просто. Один процессор – одно ядро (речь идет о бюджетных пользовательских компьютера, а не о высокопроизводительных серверах). По-настоящему многозадачность можно было реализовать на так называемых SMP-машинах, где устанавливалось как минимум два одинаковых процессора, которые подключались к общей памяти и периферийным устройства. В таких системах процессоры тесно связаны друг с другом посредством общей шины и имеют равный доступ ко всем ресурсам вычислительной системы, а также управляются одной копией операционной системы. Ранее SMP использовалась в основном на серверах, высокопроизводительных графических станциях – в общем, везде, где нужно было задействовать мощные вычислительные ресурсы. Для обычных "десктопных" целей SMP-машины не покупались – дорого, да и необходимости особой в них не было.

Итак, термином *многопроцессорный* обозначают компьютеры, имеющие несколько физически раздельных процессоров (например, серверные материнские платы часто имеют 2 или 4 сокета для подключения нескольких чипов), но управляемые одним экземпляром операционной системы (ОС).

Со временем технологии стали дешевле и стали появляться многоядерные процессоры. Многоядерным считается центральный процессор (CPU), содержащий в одном корпусе два или более вычислительных ядра на одном процессоре. С появлением многоядерности появилась некоторая путаница, поскольку есть *мультиядерные* (англ. *multi-core*) и *многоядерные* (*many-core*) процессоры. Термин *мультиядерный* обычно применяется к центральным процессорам, содержащим два и более ядра общего назначения, однако иногда используется и для цифровых сигнальных процессоров (DSP) и однокристальных систем (SoC, СнК). Под *многоядерностью* процессора понимают, что несколько ядер являются интегрированными на одну интегральную схему (изготовлены на одном кремниевом кристалле). Если же в один корпус были объединены несколько полупроводниковых кристаллов, то конструкцию называют многочиповый модуль (англ. *multi-chip module, MCM*).

Понятие *многоядерный* (англ. *many-core*) может использоваться для описания многоядерных систем, имеющих высокое количество ядер, от десятков до сотен или более. Например, именно название "многоядерный" ("many-core") использовалось Intel для вычислителей Intel MIC.

Еще большую путаницу внесла технология Hyper-Threading от Intel. В ней одно физическое ядро процессора определяется операционной системой как два логических ядра. Взять, например, процессор Intel Core i5-7200U. В нем два ядра, но каждое из них разделено на два логических ядра – говорят – на 2 потока (threads). Такая компоновка позволяет выполнять одновременно (якобы) четыре потока. Не будем вдаваться в технические особенности реализации этой технологии (об этом вы сможете прочитать в Сети), но скажем, что прибавка к производительности на процессорах с Hyper-Threading составила 30% по сравнению с процессором с таким же количеством ядер, но без HT.

Но вернемся к обсуждению многопоточности. Поскольку у нас есть несколько потоков, которые используют один и тот же контекст, нужно защитить данные от неконтролируемого одновременного доступа. Если два потока изменяют одни и те же данные, это может породить ситуацию, когда малейшее изменение в одном из потоков самым неожиданным образом повлияет на конечный результат. Представим, что у нас есть два потока, увеличивающих значение общей переменной:

```
counter = shared_counter
shared_counter = counter + 1
```

Представим, что у переменной *shared_counter* есть начальное значение 0. Представим, что потоки обрабатываю один и тот же код параллельно, см. табл. 23.1.

Таблица 23.1. Параллельная обработка кода потоками

Поток 1	**Поток 2**
`counter = shared_counter` `[counter 0]`	`counter = shared_counter` `counter 0`
`shared_counter = counter + 1` `[shared_counter = 0 + 1]`	`shared_counter = counter + 1` `[shared_counter = 0 + 1]`

В зависимости от моментов выполнения и доступности контекста может получиться результат 1 или 2. Такая ситуация называется *опасностью гонки* или *состоянием гонки*, и часто является причиной серьезных проблем в работе программы. Если ничего не предпринимать, то результатам работы такой программы мы не можем доверять.

С помощью механизмов блокировки мы можем защитить данные и программировать потоки, поскольку с их помощью мы можем убедиться, что доступ к ресурсам дается безопасным образом. Однако, неосторожное использование блокировок также может создать проблемы. Самая большая проблема возникает тогда, когда из-за неправильной организации кода два потока блокируют один и тот же ресурс и пытаются выполнить блокировку второго ресурса, который уже был заблокирован ранее. Тогда программа "зависнет". Чтобы такого не повторилось, нужно отслеживать повторное обращение, что частично решает проблему, поскольку ограничивает попытки заблокировать ресурс дважды.

Однако если потоки используются для изолированных задач (например, одна задача получает файл 1, а вторая – файл 2, или один поток загружает обновление программы, а второй – обрабатывает различные события приложения), то и использование может существенно увеличить скорость работы программы.

Многопоточность, как правило, поддерживается на уровне ядра системы. Если у машины один процессор с одним ядром, система использует технику квантования времени, когда центральный процессор переключается с одного потока на другой так быстро, что кажется, что потоки выполняются одновременно, хотя на самом деле это не так – в один момент времени выполняется какой-то один поток. В этом случае, разумеется, ни о каком приросте производительности речи идти не может.

Но все изменится, если в системе есть несколько процессоров или процессор с несколькими ядрами – тогда процессы и потоки перераспределяются между процессорами и программа выполняется быстрее.

В Python используется несколько потоков на уровне ядра и каждый из них может запустить любой из потоков уровня интерпретатора. В реализации CPython есть ограничение, делающее потоки менее пригодными для использования в разных контекстах. Все потоки, которые обращаются к объектам Python, работают с одной глобальной блокировкой. Это делается потому, что большая часть структур интерпретатора и сторонний код C небезопасны для потоков и нуждаются в защите.

Данный механизм называется GIL (*глобальная блокировка интерпретатора, Global Interpreter Lock*). Периодически в сообществе разработчиков под-

нимается тема об удалении GIL из Python, но пока процесс не сдвинулся с мертвой точки, поэтому вам придется работать с тем, что есть.

Разберемся, что такое многопоточность в Python. Если в потоках есть только код Python, использовать потоки для ускорения программы не очень правильно, поскольку GIL будет глобально сериализовать выполнение всех потоков. Но GIP работает лишь с кодом Python. На практике GIL может быть убран в расширениях C, в которых не применяются функции Python. Другими словами, несколько потоков могут производить операции ввода/вывода или выполнять код C сторонних расширений параллельно.

23.2.2. КЕЙСЫ, ПОДХОДЯЩИЕ ДЛЯ ИСПОЛЬЗОВАНИЯ МНОГОПОТОЧНОСТИ

Использовать многопоточность следует в следующих случаях:

- Когда необходимы адаптивные интерфейсы
- Делегирование работы
- Создание многопользовательских приложений

Разберем все эти случаи подробно.

АДАПТИВНЫЕ ИНТЕРФЕЙСЫ

Представим, что вы пишете какую-то программу, которой нужно скопировать файлы из одного места в другое, например, программу для резервного копирования файлов или же программу для резервного копирования базы данных. У вашей программы есть пользовательский интерфейс, и вы хотите, чтобы задача по копированию файлов/создания дампа базы данных отошла на второй план (выполнялась в фоне), а интерфейс программы показывал бы пользователю информацию о ходе выполнения задачи и давал бы возможность прекратить задачу (кнопка **Отмена**).

Хороший интерфейс позволяет пользователям работать с несколькими задачами одновременно. Достичь этого невозможно без использования потоков.

ДЕЛЕГИРОВАНИЕ РАБОТЫ

Когда работа приложения зависит от нескольких внешних ресурсов, потоки могут повысить производительность приложения. Например, у нас есть функция индексирования содержимого файлов в папке. Для этого наша программа использует какую-то внешнюю программу.

Мы можем обрабатывать файлы последовательно, выполняя нужную программу, но мы также можем создать отдельный поток для каждого конвертера и отправлять задачи по потокам. Общее время выполнения будет примерно равно времени обработки самого медленного конвертера, но оно не будет равно сумме работы всех конвертеров, как в случае с последовательным выполнением. Другими словами, если нам нужно обработать файлы размером 1, 5, 10 и 20 Мб, а время обработки равно 1 Мб/с, то при последовательном выполнении мы выполним всю задачу за 36 секунд, а при параллельном (при условии наличия четырех ядер у процессора), наша программа обработает все файлы за примерно 20 секунд.

Именно поэтому использование потоков значительно увеличивает производительность приложения и сокращает общее время выполнения.

Другой достаточно популярный кейс использования потоков – выполнение нескольких запросов к внешнему серверу. Например, вам нужно делать несколько API-запросов к какому-то веб-сервису. Если сервер находится далеко, последовательное выполнение запросов, может занять достаточно много времени – ведь вам нужно дождаться ответа от предыдущего запроса. При параллельном выполнении вы потратите гораздо меньше времени на получения ответов.

При работе с http нужно помнить, что при выполнении запроса максимальное время тратиться на чтение из TCP-сокета. Данная операция блокирует I/O, из-за этого Python снимает GIL при выполнении C-функции *recv()*, что позволяет существенно улучшить производительность приложения.

МНОГОПОЛЬЗОВАТЕЛЬСКИЕ ПРИЛОЖЕНИЯ

Рассмотрим обратный пример – мы разрабатываем приложение, не обращающееся к внешнему http-сервису, а которое само является веб-сервером. Веб-сервер должен принять запрос пользователя, поставить его в новый поток, а затем ожидать новых запросов. Отдельный поток на пользовате-

ля существенно упрощает работу разработчика. Конечно, нужно помнить о блокировках, если это запрос на запись. С запросами на чтение все гораздо проще.

23.3. Практика: создание многопоточного приложения

23.3.1. МОДУЛЬ THREAD

Запуск нескольких потоков аналогичен одновременному запуску нескольких разных программ, но со следующими преимуществами:

- Несколько потоков в рамках процесса используют одно и то же пространство данных с основным потоком и поэтому могут обмениваться информацией или взаимодействовать друг с другом более легко, чем если бы они были отдельными процессами.
- Потоки иногда называют легковесными процессами, и они не требуют больших затрат памяти; они дешевле процессов.

У потока есть начало, последовательность выполнения и завершение. У него есть указатель инструкции, который отслеживает, где в его контексте он выполняется в данный момент. Поток может быть легко прерван. Поток можно временно приостановить, пока работают другие потоки – это называется уступкой (yield).

Разберемся, как начать новый поток. Для этого нам нужно вызвать следующий метод, находящийся в модуле *thread*:

```
thread.start_new_thread ( function, args[, kwargs] )
```

Данный метод позволяет создать потоки, как в Linux, так и в Windows. Вызов метода немедленно возвращается, запуская дочерний поток. В качестве кода потока будет использована функция, заданная первым аргументом. Остальные аргументы передаются вызываемой функции. Когда функция завершает работу, то завершает работу и поток.

Здесь *args* – это набор аргументов, передаваемых функции. Если функции не нужно передавать аргументы, используйте пустой кортеж. Последний параметр – необязательный. Это словарь аргументов ключевых слов.

Рассмотрим полноценный пример (лист. 23.1).

Листинг 23.1. Запуск потока

```
#!/usr/bin/python

import thread
import time

# Функция для потока
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Создаем 2 потока
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Ошибка: не могу создать поток"

while 1:
   pass
```

Код из листинга 23.1 породит следующий вывод:

```
Thread-1: Mon Oct 04 08:42:17 2021
Thread-1: Mon Oct 04 08:42:19 2021
Thread-2: Mon Oct 04 08:42:19 2021
Thread-1: Mon Oct 04 08:42:21 2021
Thread-2: Mon Oct 04 08:42:23 2021
Thread-1: Mon Oct 04 08:42:23 2021
Thread-1: Mon Oct 04 08:42:25 2021
Thread-2: Mon Oct 04 08:42:27 2021
Thread-2: Mon Oct 04 08:42:31 2021
Thread-2: Mon Oct 04 08:42:35 2021
```

Примечание. Вывод будет у вас отличаться – в зависимости от даты выполнения и настроек локали.

Хотя модуль *thread* очень эффективен для низкоуровневой обработки потоков, он очень ограничен по сравнению с более новым модулем *threading*.

23.3.2. МОДУЛЬ THREADING

Более новый модуль *threading*, впервые появившийся в в Python 2.4, обеспечивает гораздо более мощную и высокоуровневую поддержку потоков, чем модуль *thread*, рассмотренный в предыдущем разделе.

Модуль *threading* предоставляет все методы модуля *thread* и некоторые дополнительные методы:

- *threading.activeCount()* – возвращает количество активных объектов потока;
- *threading.currentThread()* – возвращает количество объектов потока в элементе управления потоком вызывающего объекта;
- *threading.enumerate()* – возвращает список всех активных в данный момент объектов потока.

В дополнение к методам, в модуле *threading* есть класс Thread, реализующий потоки. Класс Thread предоставляет следующие методы:

- *run()* – является точкой входа для потока;
- *start()* – запускает поток, вызывая метод run();
- *join([время])* – ожидает завершения потоков;
- *isAlive()* – проверяет, выполняется ли еще поток;
- *getName()* – возвращает имя потока;
- *setName()* – устанавливает имя потока.

Рассмотрим, как можно создать поток, используя метод *threading*. Для этого нам нужно сделать следующее:

1. Определить новый подкласс класса Thread.
2. Переопределить метод __init(self[,args])__ для добавления дополнительных аргументов
3. Переопределить метод run(self[,args]) для реализации того, что должен сделать поток после запуска.

Создав новый подкласс Thread, вы можете создать его экземпляр, а затем запустить новый поток, вызвав start(), который, в свою очередь, вызывает метод *run()*. Полный пример показан в лист. 23.2.

Листинг 23.2. Пример создания потоков с использованием модуля threading

```
#!/usr/bin/python

import threading
import time

exitFlag = 0

class myThread (threading.Thread):
   def __init__(self, threadID, name, counter):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print "Запу ск" + self.name
      print_time(self.name, 5, self.counter)
      print "Выход из " + self.name

def print_time(threadName, counter, delay):
   while counter:
      if exitFlag:
         threadName.exit()
      time.sleep(delay)
      print "%s: %s" % (threadName, time.ctime(time.time()))
      counter -= 1

# Создание новых потоков
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Запу скпотоков
```

```
thread1.start()
thread2.start()

print "Выход из главного потока"
```

Вывод программы будет следующий:

```
Запуск Thread-1
Запуск Thread-2
Выход из главного потока
Thread-1: Mon Oct 04 09:10:03 2021
Thread-1: Mon Oct 04 09:10:04 2021
Thread-2: Mon Oct 04 09:10:04 2021
Thread-1: Mon Oct 04 09:10:05 2021
Thread-1: Mon Oct 04 09:10:06 2021
Thread-2: Mon Oct 04 09:10:06 2021
Thread-1: Mon Oct 04 09:10:07 2021
Выход из Thread-1
Thread-2: Mon Oct 04 09:10:08 2021
Thread-2: Mon Oct 04 09:10:10 2021
Thread-2: Mon Oct 04 09:10:12 2021
Выход из Thread-2
```

23.3.3. СИНХРОНИЗАЦИЯ ПОТОКОВ

Модуль *threading* содержит простой в реализации механизм блокировки, позволяющий синхронизировать потоки. Новая блокировка создается путем вызова метода *Lock()*, который возвращает новую блокировку.

Метод *acquire(blocking)* нового объекта блокировки используется для принудительного выполнения потоков синхронно. Необязательный параметр *blocking* позволяет вам контролировать, ожидает ли поток получения блокировки.

Если *blocking* равен 0, поток немедленно завершается со значением 0, если блокировка не может быть получена и 1, если блокировка была получена. Если *blocking* равен 1, поток будет заблокирован в ожидании снятия блокировки.

Метод *release()* используется для снятия блокировки, когда она больше не нужна. В листинге 23.3 приведен пример работы с блокировками.

Листинг 23.3. Работаем с блокировками

```
#!/usr/bin/python

import threading
import time

class myThread (threading.Thread):
   def __init__(self, threadID, name, counter):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print "Запуск " + self.name
      # Получаем блокировку для синхронизации потока
      threadLock.acquire()
      print_time(self.name, self.counter, 3)
      # Снимаем блокировку для освобождения нового потока
      threadLock.release()

def print_time(threadName, delay, counter):
   while counter:
      time.sleep(delay)
      print "%s: %s" % (threadName, time.ctime(time.time()))
      counter -= 1

threadLock = threading.Lock()
threads = []

# Создаем новые потоки
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Запускаем новые потоки
thread1.start()
thread2.start()

# Добавляем потоки в список потоков
threads.append(thread1)
threads.append(thread2)

# Ждем завершения всех потоков
for t in threads:
    t.join()
print "Выход из главного потока"
```

Вывод будет следующим:

```
Запуск Thread-1
Запуск Thread-2
Thread-1: Mon Oct 04 09:11:28 2021
Thread-1: Mon Oct 04 09:11:29 2021
Thread-1: Mon Oct 04 09:11:30 2021
Thread-2: Mon Oct 04 09:11:32 2021
Thread-2: Mon Oct 04 09:11:34 2021
Thread-2: Mon Oct 04 09:11:36 2021
Выход из главного потока
```

23.3.4. МНОГОПОТОЧНАЯ ПРИОРИТЕТНАЯ ОЧЕРЕДЬ

Модуль Queue позволяет вам создавать новые объект очереди, позволяющий хранить определенное число элементов. Следующие методы используются для управления очередью:

- *get()* – удаляет и возвращает элемент из очереди;
- *put()* – добавляет элемент в очередь;
- *qsize()* – возвращает количество элементов, которые в настоящее время находятся в очереди;
- *empty()* – возвращает *True*, если очередь пуста; в противном случае – *False*;
- *full()* – возвращает *True*, если очередь заполнена; в противном случае – *False*.

Рассмотрим пример (лист. 23.4).

Листинг 23.4. Работа с очередью

```
#!/usr/bin/python

import Queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
```

```
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print "Запуск " + self.name
        process_data(self.name, self.q)
        print "Выход из " + self.name

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
            if not workQueue.empty():
                data = q.get()
                queueLock.release()
                print "%s обрабатывает %s" % (threadName, data)
            else:
                queueLock.release()
            time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["Марк", "Денис", "Евгения", "Валерия", "Андрей"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1

# Create new threads
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# Заполняем очередь элементами из списка имен nameList
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# Ждем пока освободится очередь
while not workQueue.empty():
    pass

# Уведомляем потоки, что настало время завершения
exitFlag = 1

# Ждем пока завершат работу все потоки
```

```
for t in threads:
   t.join()
print "Выход из основного потока"
```

Вывод скрипта будет следующим:

```
Запуск Thread-1
Запуск Thread-2
Запуск Thread-3
Thread-1 обрабатывает Марк
Thread-2 обрабатывает Денис
Thread-3 обрабатывает Евгения
Thread-1 обрабатывает Валерия
Thread-2 обрабатывает Андрей
Выход из Thread-3
Выход из Thread-1
Выход из Thread-2
Выход из основного потока
```

Как видно, три потока поочередно обрабатывают элементы из списка имен.

23.4. Практический пример: многопотоковый сетевой сервер

В этом разделе будет показано, как создать сетевое приложение, обслуживающее нескольких клиентов. Возможно, оно станет основой для вашего собственного проекта. Начнем с разработки сервера – программа, которая будет "слушать" сокет и принимать соединения от клиентов. Наш сервер будет слушать порт с номером 8888. Порты с номерами 80, 448, 8080, 8000, 8081, 3128 использовать не рекомендуется, поскольку они могут быть заняты веб-сервером или прокси-сервером. Чтобы сервер слушал порт, нам нужно выполнить два вызова:

```
server.bind((LOCALHOST, PORT))
server.listen(1)
```

Здесь LOCALHOST – это имя локального компьютера (с адресом 127.0.0.1), а PORT – нужный нам порт. Метод *bind()* связывает наш сервер с определенным портом. А вот метод *listen()* запускает прослушку этого порта.

Листинг 23.5. Классический сервер, обслуживающий одного клиента

```
import socket
LOCALHOST = "127.0.0.1"
PORT = 8888
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((LOCALHOST, PORT))
server.listen(1)
print("Ждем запрос клиента...")
while True:
    clientConnection,clientAddress = server.accept()
    print("Подключился клиент :" , clientAddress)
    data = clientConnection.recv(1024)
    print("Получено от клиента :" , data.decode())
    clientConnection.send(bytes("Hello",'UTF-8'))
    clientConnection.close()
```

Наш сервер ожидает запрос клиента. Как только клиент подключится, сервер примет соединение (метод *accept()*), получит данные от клиента (метод *recv()*) и отправит ему в отчет строчку Hello (метод *send()*).

Программа-клиент приведена в лист. 23.6.

Листинг 23.6. Программа-клиент

```
import socket
SERVER = "127.0.0.1"
PORT = 8888
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((SERVER, PORT))
client.sendall(bytes("Hello, server!!!",'UTF-8'))
data =  client.recv(1024)
print(data.decode())
client.close()
```

Клиент работает просто – он подключается к серверу и отправляет строчку Hello, server!!!. Кодировка UTF-8 (вы можете использовать кириллицу).

Теперь усложним задачу – напишем многопотоковый сетевой сервер. Он будет запускать отдельный поток для обработки каждого нового клиента – подобно тому, как работают "настоящие" серверы. Для реализации нашей

программ мы будем использовать модули *socket* и *threading*. Код программы-сервера приведен в лист. 23.7.

Листинг 23.7. Код многопотокового сервера

```
import socket, threading
class ClientThread(threading.Thread):
    def __init__(self,clientAddress,clientsocket):
        threading.Thread.__init__(self)
        self.csocket = clientsocket
        print ("Новое соединение: ", clientAddress)
    def run(self):
        print ("Адрес клиента : ", clientAddress)
        #self.csocket.send(bytes("Hi, This is from
Server..",'utf-8'))
        msg = ''
        while True:
            data = self.csocket.recv(2048)
            msg = data.decode()
            if msg=='bye':
              break
            print ("от клиента ", msg)
            self.csocket.send(bytes(msg,'UTF-8'))
        print ("Клиент ", clientAddress , " отключился...")
LOCALHOST = "127.0.0.1"
PORT = 8888
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((LOCALHOST, PORT))
print("Ждем запросы...")
while True:
    server.listen(1)
    clientsock, clientAddress = server.accept()
    newthread = ClientThread(clientAddress, clientsock)
    newthread.start()
```

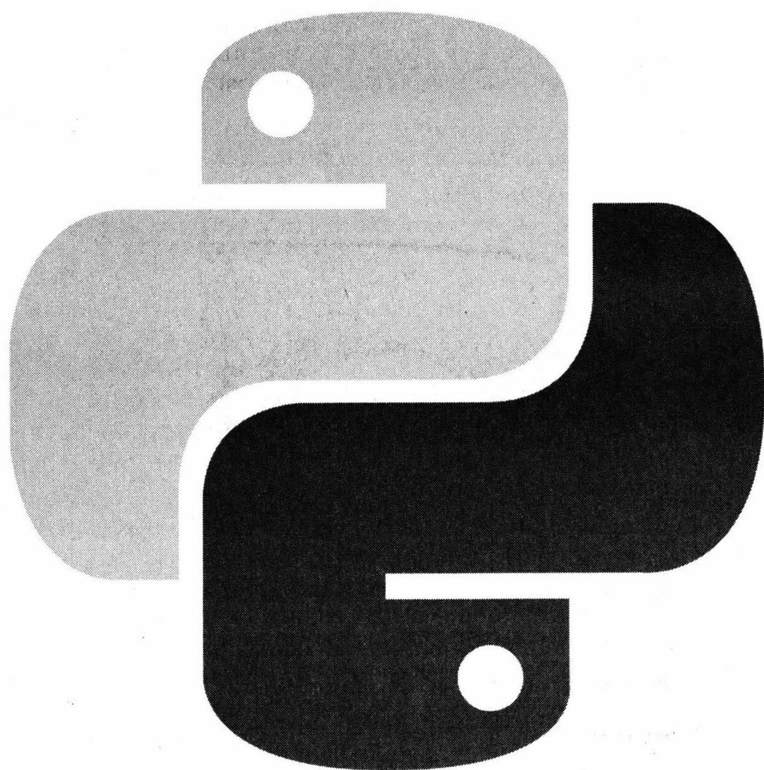
Сначала мы определяем объект ClientThread класса threading.Thread. Мы переопределяем метод __init__ – инициализация и вывод информации о клиенте. Затем мы переопределяем метод *run()* – это и будет наш метод обработки соединения с клиентом. Мы просто принимаем сообщения от клиента, пока не увидим сообщение *bye*. Как только клиент передает там это сообщение, мы прерываем обработку клиента.

Программу-клиент (лист. 23.8) также нужно немного переделать. Технически она мало чем отличается от первой версии, просто добавлена реакция на ключевое слово *bye*.

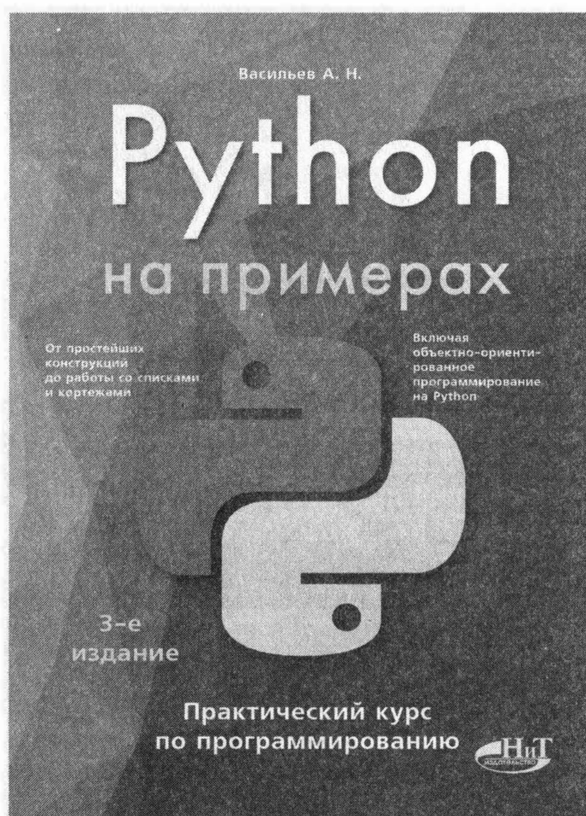
Листинг 23.8. Новый клиент

```
import socket
SERVER = "127.0.0.1"
PORT = 8888
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((SERVER, PORT))
client.sendall(bytes("This is from Client",'UTF-8'))
while True:
  in_data =  client.recv(1024)
  print("От сервера :" ,in_data.decode())
  out_data = input()
  client.sendall(bytes(out_data,'UTF-8'))
  if out_data=='bye':
  break
client.close()
```

Как запускать все это? Первым делом откройте командную строку (или терминал) и запустите сервер (python server.py), а затем откройте несколько терминалов и запустите в каждом из них по клиенту (python client.py). А затем наблюдайте за волшебством! Вы увидите, что наш сервер обрабатывает параллельно каждого клиента.



"Издательство Наука и Техника" рекомендует:



Python на примерах. Практический курс по программированию. 3-е издание. — СПб.: Наука и Техника. — 432 с., ил.

В этой книге решать будем две задачи, одна из которых приоритетная, а вторая, хотя и вспомогательная, но достаточно важная. Наша основная задача, конечно же, изучение синтаксиса языка программирования Python. Параллельно мы будем осваивать программирование как таковое, явно или неявно принимая во внимание, что соответствующие алгоритмы предполагается реализовывать на языке Python.

Большинство авторов книг в своих трудах рассматривают теоретические основы языка и уделяют основное внимание базовому синтаксису языка, не рассматривая при этом практическую сторону его применения. Эта же книга старается восполнить недостаток практического материала, содержит множество примеров с комментариями, которые вы сможете использовать в качестве основы своих программных решений, изучения Python.

Материал книги излагается последовательно и сопровождается большим количеством наглядных примеров, разноплановых практических задач и детальным разбором их решений

Кольцов Дмитрий Михайлович

PYTHON
полное руководство

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО "Издательство Наука и Техника"
ОГРН 1217800116247, ИНН 7811763020, КПП 781101001
192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107, лит. Б, пом. 1-Н
Подписано в печать 27.10.2021. Формат 70x100 1/16.
Бумага газетная. Печать офсетная. Объем 30 п. л.
Тираж 1500. Заказ № 1031.



Отпечатано в АО «Можайский полиграфический комбинат»
143200, Россия, г. Можайск, ул. Мира, 93.
www.oaompk.ru, тел.: (495) 748-04-67, (49638) 20-685

PYTHON

Полное руководство

Кольцов Д. М.

Эта книга поможет вам освоить язык программирования Python практически с нуля, поэтапно, от простого к сложному. Первая часть книги посвящена базовым основам языка: переменные и типы данных, операторы, циклы и условные операторы, математические функции, кортежи, множества и словари, итераторы и генераторы, модули и пакеты, а также многое другое. Во второй части книги перейдем к более сложным вещам в Python: объектно-ориентированное программирование, метапрограммирование, многопоточность и масштабирование.

Отдельное внимание будет уделено документированию своего проекта в Python, контролю и оптимизации кода. Теоретическая часть книги сопровождается практическими примерами, позволяющими на практике осваивать полученные теоретические знания.

Книга будет полезна как начинающим, так и тем, кто хочет улучшить свои навыки программирования на Python.


ISBN 978-5-94387-270-9
9 78- 5- 94387- 270- 9

"Издательство Наука и Техника"
г. Санкт-Петербург

Для заказа книг:
(812) 412-70-26
e-mail: nitmail@nit.com.ru
www.nit.com.ru



www.nit.com.ru